An Algebra for Features and Feature Composition

Sven Apel (University of Passau)

Christian Lengauer (University of Passau)

Don Batory (UT Austin)

Bernhard Möller (University of Augsburg)

Christian Kästner (University of Magdeburg)

Passau Technical Report MIP-0706 (on the Web)

Programming Group

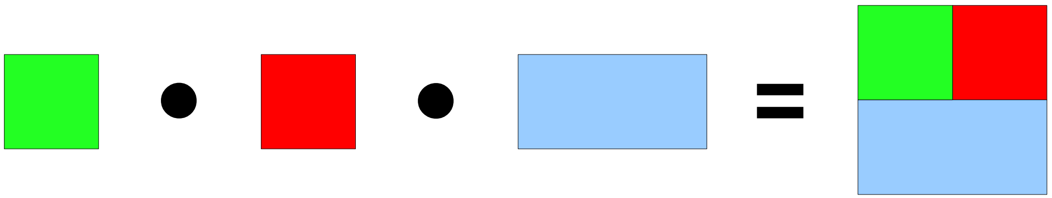
UNIVERSITÄT PASSAU

Feature-Oriented Software Development

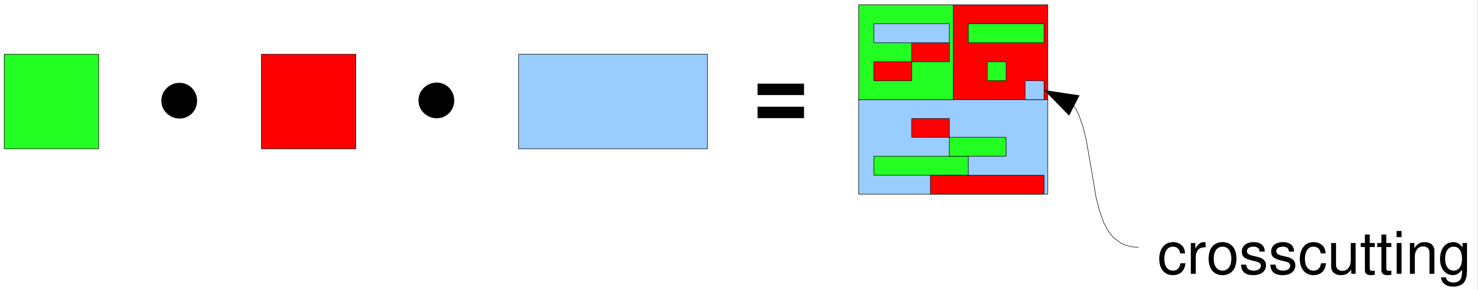
- What is a feature?
  - Increment in program functionality
  - Maps to a requirement
  - Used to distinguish different programs
- Idea: represent features explicitly in design and code
  - Each feature is encapsulated in a module
  - A feature refines a (possibly empty) program
  - A final program is composed of a series of features

Feature Composition

- A simple model of feature composition (•)



- The reality is more complicated



Diverse Implementation Approaches

- Components
- Collaboration-based design
- Aspect-oriented programming
- Generative programming
- Frames
- ...

➔ What are the essence and the principles of features and feature composition?

Our Approach

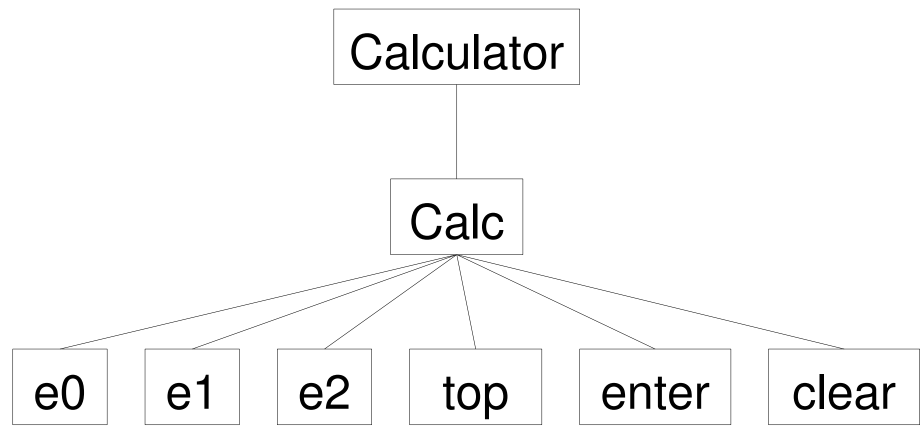
Programming

Group

UNIVERSITÄT PASSAU

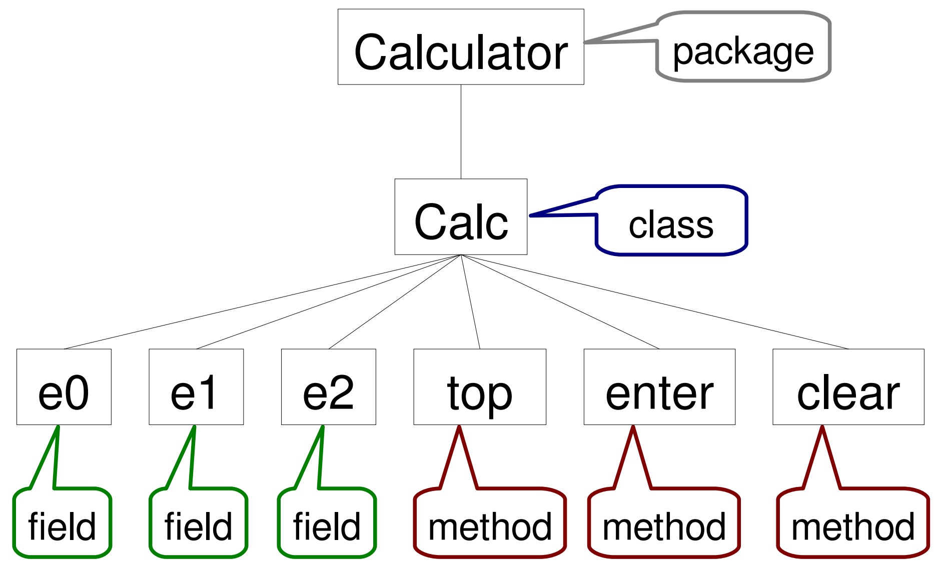
Features are Trees

```
class Calc {
  int e0 = 0, e1 = 0, e2 = 0;
  void enter(int val) {
    e2 = e1; e1 = e0; e0 = val;
  }
  void clear() {
    e0 = e1 = e2 = 0;
  }
  String top() {
    return String.valueOf(e0);
  }
}
```



Nodes Have Different Types

```
class Calc {
  int e0 = 0, e1 = 0, e2 = 0;
  void enter(int val) {
    e2 = e1; e1 = e0; e0 = val;
  }
  void clear() {
    e0 = e1 = e2 = 0;
  }
  String top() {
    return String.valueOf(e0);
  }
}
```



Feature Composition is Tree Superimposition

```
package Calculator;
class Calc {
  void add() {
    e0 = e1 + e0; e1 = e2;
  }
}
```

feature: Add

•

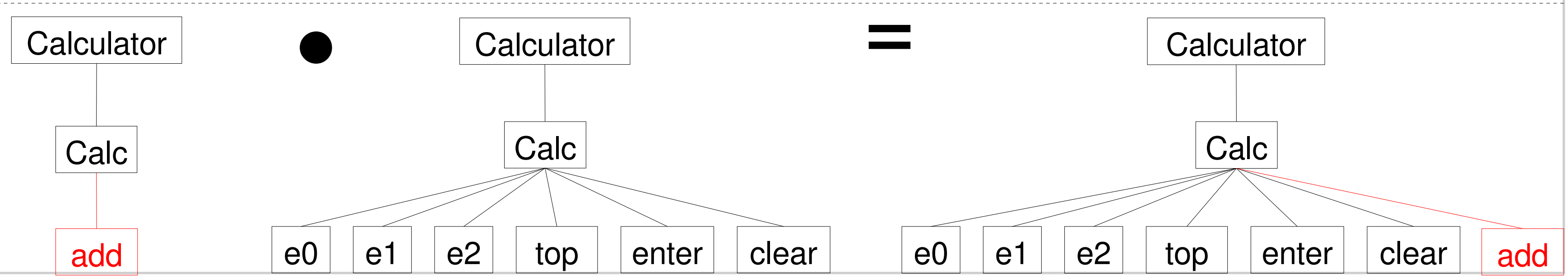
```
package Calculator;
class Calc {
  int e0 = 0, e1 = 0, e2 = 0;
  void enter(int val) {
    e2 = e1; e1 = e0; e0 = val;
  }
  void clear() {
    e0 = e1 = e2 = 0;
  }
  String top() {
    return String.valueOf(e0);
  }
}
```

feature: CalcBase

=

```
package Calculator;
class Calc {
  int e0 = 0, e1 = 0, e2 = 0;
  void enter(int val) {
    e2 = e1; e1 = e0; e0 = val;
  }
  void clear() {
    e0 = e1 = e2 = 0;
  }
  String top() {
    return String.valueOf(e0);
  }
  void add() {
    e0 = e1 + e0; e1 = e2;
  }
}
```

feature: CalcAdd



Feature Composition

- Recursive composition of the elements of a feature
  - *package* • *package* → *package* (also for subpackages)
  - *class* • *class* → *class* (also for inner classes)
  - *interface* • *interface* → *interface* (also for inner interfaces)
  - *method* • *method* → ?
  - *field* • *field* → ?

What About Fields and Methods?

- If the same signature, there are two options:
  1. Consider these compositions errors
  2. Allow these compositions under some circumstances
     - *method* • *method* → *method* if one method calls **`original`**
     - *field* • *field* → *field* if one field is abstract (no value assigned)

Composition of Fields and Methods

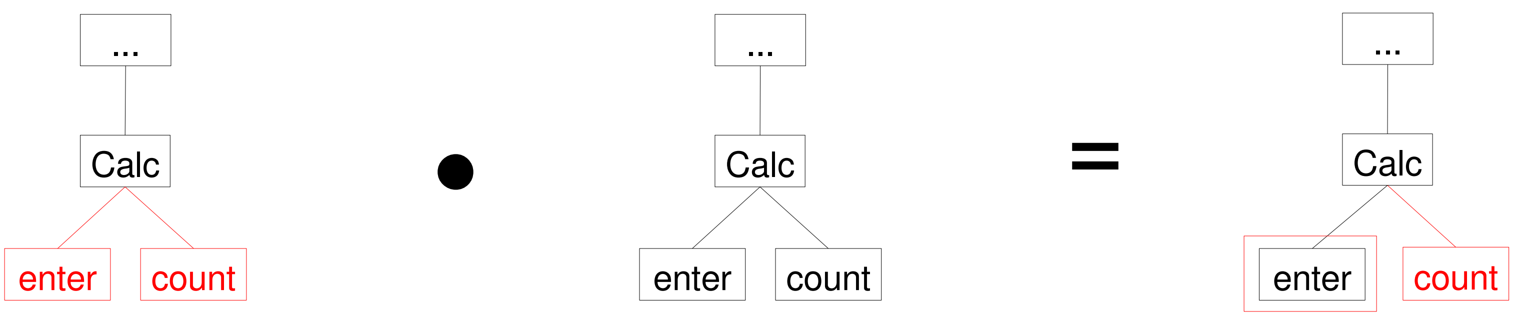
```
class Calc {
  int count = 0;
  void enter(int val) {
    original.enter(val);
    count++;
  }
}
```

•

```
class Calc {
  int count;
  void enter(int val) {
    e2 = e1; e1 = e0; e0 = val;
  }
}
```

=

```
class Calc {
  int count = 0;
  void enter(int val) {
    e2 = e1; e1 = e0; e0 = val;
    count++;
  }
}
```



A Feature Algebra

Programming

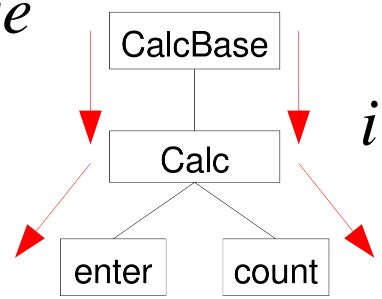
Group

UNIVERSITÄT
PASSAU

Paths and Introductions

- A feature is represented as a tree
- Each path in the tree, starting at the root, is called an introduction

$$i_1 = CalcBase$$



$$i_2 = CalcBase.Calc$$

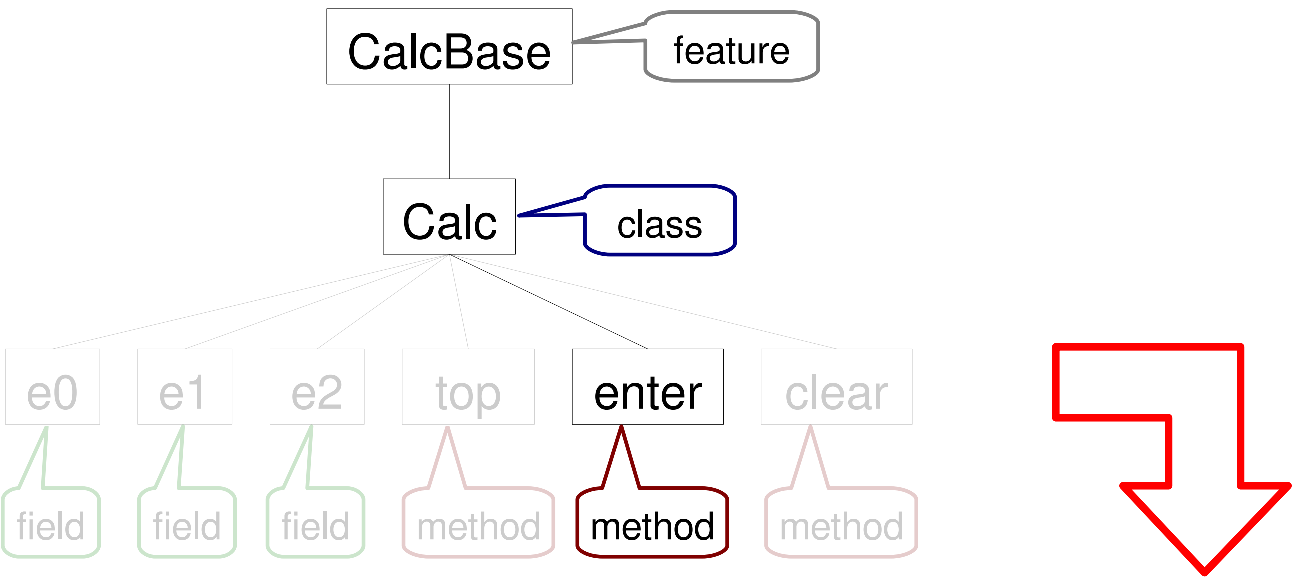
$$i_3 = CalcBase.Calc.enter$$

$$i_4 = CalcBase.Calc.count$$

- The hierarchical structure is encoded in the identifiers

Introductions Encode Type Information

- Similarly to trees that represent features, an introduction encodes type information in its identifier



$$i = CalcBase^{feature}.Calc^{class}.enter^{method}$$

Features and Introductions

- A feature is a sum of introductions ($\oplus$)

$$
\begin{aligned}
F = & \, CalcBase \\
& \oplus CalcBase.Calc \\
& \oplus CalcBase.Calc.e0 \\
& \oplus CalcBase.Calc.e1 \\
& \oplus CalcBase.Calc.e2 \\
& \oplus CalcBase.Calc.enter \\
& \oplus CalcBase.Calc.clear \\
& \oplus CalcBase.Calc.top
\end{aligned}
$$

A compiler can evaluate this meta-expression and synthesize the according code

the set of summands is prefix-closed

Introduction Sum

- Introduction sum over the set of introductions I forms an **idempotent non-commutative monoid (I, $\oplus$)**
  - Closure: composing introductions creates new introductions
  - Associativity: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
  - Identity: $\xi \oplus a = a \oplus \xi = a$ (ξ is the empty introduction)
  - Non-commutativity: $a \oplus b \neq b \oplus a$
  - Distant idempotence: $a \oplus b \oplus a = b \oplus a$
    - implies direct idempotence: $a \oplus a = a$

Feature Composition is Introduction Sum

- Features are composed by introduction sum

$$F = \underbrace{i_1 \oplus \ldots \oplus i_{m_0}}_{F_0} \oplus \ldots \oplus \underbrace{i_1 \oplus \ldots \oplus i_{m_n}}_{F_n}$$

Semantics of Introduction Sum

- An introduction adds a new path to a tree
- Introductions are composed recursively
- Nodes with the same name and type are merged
  - Node wrapping
  - Subtree merging

Applications of Introductions

- Node wrapping
  - Extend methods via overriding
  - Shield field accesses
  - Declare new superclasses and interfaces
- Subtree merging
  - Add new packages, classes, interfaces, methods, fields

Modification – A Further Element of a Feature

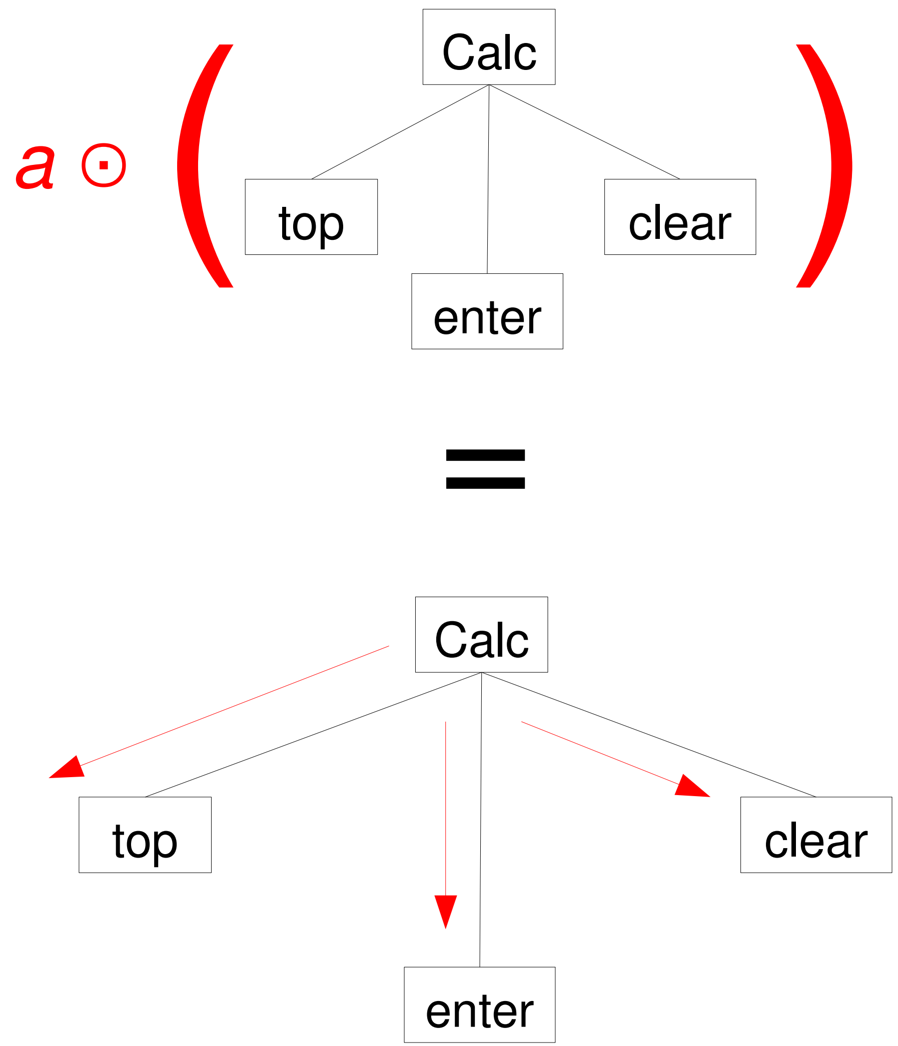
Programming Group

UNIVERSITÄT PASSAU

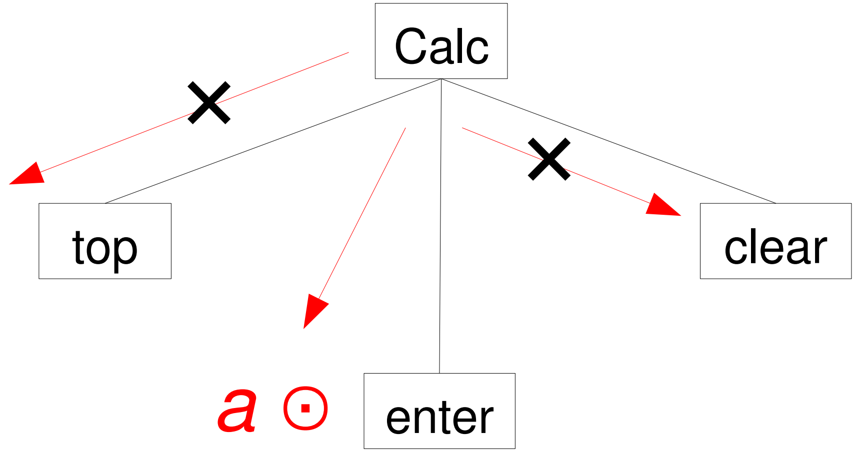
Modifications

- Meta-level entities / meta-programming constructs
- Quantify over introductions
- Apply changes to a feature declaratively
- General form of modifications
  - Where does something happen?
  - What happens?
  - (What conditions must hold?)

Modifications are Tree Walks



Where is a Modification Applied?



a = *.enter

Effects of Modifications

- Same as introduction sum
  - Add new children to a node
  - Wrapping of nodes

So What is the Difference Between Introductions and Modifications?

- Both can wrap nodes and add subtrees
- The difference between both is how this is expressed
  - Where does something happen?
  - What happens?

Back to our Feature Algebra

Programming

Group

Operations on Modifications

- Modifications can be added (just like introductions)

$$\oplus : A \times A \rightarrow A$$

- Modifications can be composed like functions

$$\odot : A \times A \rightarrow A$$

- Modifications can be applied to introductions

$$\odot : A \times I \rightarrow I$$

Modification Sum

- $\oplus : A \times A \rightarrow A$
- Modifications consist of two parts
  - Where does something happen?
    - The sum modifies the union of join points
  - What happens?
    - Disjoint sets of join points
      - Wrappers and subtree introductions can be merged to one set
    - Overlapping join points
      - Wrappers are nested based on the summation order
      - Subtrees are merged based on the summation order

Modification Application

- $\odot : A \times I \to I$

- Modifications are applied to sums of introductions

$$a \odot (i \oplus j) = (a \odot i) \oplus (a \odot j) \quad \text{with} \quad a \in A \quad \text{and} \quad i, j \in I$$

Modification Composition

- $\odot : A \times A \rightarrow A$

- Modification composition is successive modification application

$$(a \odot b) \odot i = a \odot (b \odot i) \quad \text{with} \quad a, b \in A \quad \text{and} \quad i \in I$$

Algebraic Properties of Modifications

- Modifications and their operations form a **binoid** $(A, \oplus, \odot)$
  - $(A, \oplus)$ with $\oplus: A \times A \to A$ is an **idempotent monoid**
    - Associativity: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
    - Identity: $\zeta \oplus a = a \oplus \zeta = a$ (ζ is the empty modification)
    - Non-commutativity: $a \oplus b \neq b \oplus a$
    - Distant idempotence: $a \oplus b \oplus a = b \oplus a$
      - implies direct idempotence: $a \oplus a = a$
  - $(A, \odot)$ with $\odot: A \times A \to A$ is a **monoid**
    - Associativity: $(a \odot b) \odot c = a \odot (b \odot c)$
    - Identity: $\zeta \odot a = a \odot \zeta = a$ (ζ is the empty modification)
    - Non-commutativity: $a \odot b \neq b \odot a$

Algebraic Properties of Features

(I, $\oplus$) is a **semimodul** over the **binoid** (A, $\oplus$, $\odot$)

$$a \odot (i \oplus j) = (a \odot i) \oplus (a \odot j) \quad \text{with} \quad a \in A \quad \text{and} \quad i, j \in I$$

$$(a \oplus b) \odot i = (a \odot i) \oplus (b \odot i) \quad \text{with} \quad a, b \in A \quad \text{and} \quad i \in I$$

$$(a \odot b) \odot i = a \odot (b \odot i) \quad \text{with} \quad a, b \in A \quad \text{and} \quad i \in I$$

Quarks

Programming

Group

UNIVERSITÄT
PASSAU

Two Interpretations of Modifications

- **Local modifications** (*a*) affect only introductions of the features applied before

$$F_3 \bullet F_2 \bullet F_1 \bullet X = i_3 \oplus a_3 \odot (i_2 \oplus a_2 \odot (i_1 \oplus a_1 \odot X))$$

- **Global modifications** (*g*) may affect all introductions of a series of features

$$F_3 \bullet F_2 \bullet F_1 \bullet X = (g_3 \odot g_2 \odot g_1) \odot (i_3 \oplus i_2 \oplus i_1 \oplus X)$$

Quarks

- A feature consists of introductions and local / global modifications of different orders
- Composition of features is not obvious ➔ ***quarks***
- A quark is a triple and represents a feature

$$q = \langle g, i, a \rangle$$

Quark Composition

$$\langle g_1, i_1, a_1\rangle\langle g_2, i_2, a_2\rangle = \langle g_1 \odot g_2, i_1 \oplus (a_1 \odot i_2), a_1 \odot a_2\rangle$$

Conclusions