An Overview of the *g*DEEP Calculus

Sven Apel[†] and DeLesley Hutchins[‡]

[†] Department of Informatics and Mathematics, University of Passau
apel@uni-passau.de

[‡] School of Informatics, University of Edinburgh
d.s.hutchins@sms.ed.ac.uk

Technical Report, Number MIP-0712
Department of Informatics and Mathematics
University of Passau, Germany
November 2007

An Overview of the *g*DEEP Calculus

Sven Apel[†] and DeLesley Hutchins[‡]

[†] Department of Informatics and Mathematics, University of Passau
apel@uni-passau.de
[‡] School of Informatics, University of Edinburgh
d.s.hutchins@sms.ed.ac.uk

Abstract. The goal of *Feature-oriented Programming (FOP)* is to modularize software systems in terms of features. A *feature* is an increment in functionality and refines the content of other features. A software system typically consists of a collection of different kinds of software artifacts, e.g. source code, build scripts, documentation, design documents, and performance profiles. We and others have noticed a *principle of uniformity*, which dictates that when composing features, all software artifacts can actually be refined in a uniform way, regardless of what they represent. Previous work did not take advantage of this uniformity; each kind of software artifact used a separate tool for composition, developed from scratch. We present *g*DEEP, a core calculus for features and feature composition which is language-independent; it can be used to compose features containing any kinds of artifact. This calculus allows us to define general algorithms for feature refinement, composition, and validation. We provide the formal syntax, operational semantics, and type system of *g*DEEP and explain how different kinds of software artifacts, including Java, Bali, and XML files, can be represented. A prototype tool and three case studies demonstrate the practicality of our approach.

1 Introduction

The goal of *Feature-oriented Programming (FOP)* is to modularize software systems in terms of features [1,2]. A *feature* reflects a stakeholder's requirement and is an increment in program functionality. *AHEAD (Algebraic Hierarchical Equations for Application Design)* is an architectural model for FOP and a framework for large-scale program synthesis based on features [2].

The idea behind AHEAD is to unify several approaches of FOP and scale them to programming in the large. First, AHEAD generalizes the operations that are performed when a feature is composed with a base program. A feature encapsulates a *program refinement*, which is a set of changes to a base program. Such changes include the introduction of new program elements and the modification of existing program elements. Second, AHEAD scales the idea of program refinement to arbitrary kinds of software artifacts. A feature typically includes changes to not only the source code, but also to other supporting documents, e.g., HTML documentation, ANT build scripts, performance models, and design documents.

The unification and scaling of refinement is captured by the principle of uniformity: *features are implemented by a diverse selection of software artifacts, and any kind of*

software artifact can be the subject of subsequent refinement [2]. We call a feature consisting of different kinds of artifacts a *multi-representation feature*.

While the principle of uniformity captures the philosophy behind AHEAD and guides us when reasoning about feature-based program synthesis [3], it is rather abstract. During our work on FOP [4,5,6,7,8,9,10] we realized that the formal foundation of AHEAD and the principle of uniformity alone do not provide a sufficient basis for efficient software development. When we were implementing tools for composing programs from source code artifacts (e.g., *FeatureC++* [10] and *ARJ* [6]), we observed that we needed tools for more and more kinds of artifacts. We noticed that, despite the uniform process of program refinement, every time we introduced a new kind of software artifact, we had to introduce new language constructs and build new tools from scratch.

We realized that although program refinement is similar for all artifact types, we had no way to express and reason about this similarity. We need a formal mathematical model of the principle of uniformity in order to answer several important questions: What is the essence of program refinement? What properties are mandatory for software artifacts to be refined? What is common to all artifact languages and what are the differences? A theory of features and feature composition would help us answer these questions.

We propose *g*DEEP, a core calculus for feature composition, as the backbone of such a theory. The *g*DEEP calculus has several benefits:

1. It enables us to describe and analyze the properties and procedures of program refinement and feature composition in a formal way.
2. Software artifacts of different types (e.g., Java, C++, HTML, XML) can be plugged into the calculus and treated equally by the algorithms for feature refinement, composition, and validation. The algorithms can be expressed in a uniform and language-independent way.
3. Tools that implement the algorithms operate directly on a concrete representation of *g*DEEP. This way, a tool implemented for a specific problem can be reused for various types of artifacts.

*g*DEEP generalizes and scales previous work on a formal foundation for FOP [4] in order to capture the principle of uniformity. It is an alternative to related approaches that rely on algebra [11,12,13]. We present the syntax, operational semantics, and type system of *g*DEEP and explain how different artifact languages can be plugged into it, including Java, Bali, and XML. Furthermore, we share our experience with a prototype tool that implements feature composition on top of *g*DEEP.

2 Feature-Oriented Programming

2.1 Features and Feature Composition

A feature refines the content of a base program by either adding new elements, or by modifying and extending existing elements. Mathematically, we treat features as functions that transform their input in a well-defined way. Features are composed together

to synthesize individual programs, and such composition is static (i.e., done at compile-time).

A feature composition starts off with a base program, and then applies a sequence of features to modify and extend it. The order in which features are applied is important; earlier features in the sequence may add elements that are refined by later features. A sequence of features along with a base program forms a layered design, as illustrated in Figure 1.

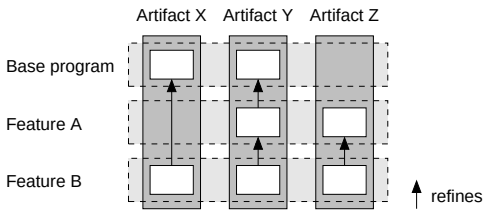

Fig. 1. A layered feature-oriented design.

2.2 AHEAD

AHEAD is an architectural model for large-scale feature-based program synthesis [2]. Features consist not only of source code, but also of other kinds of software artifacts, e.g., documentation, test cases, design documents, makefiles, performance profiles, mathematical models, etc. Every kind of software artifact that is part of a feature can be refined by subsequent features.

With AHEAD, each feature is implemented by a *containment hierarchy*, which is a directory that maintains a subdirectory structure organizing the feature's artifacts. Composing features means composing containment hierarchies and, to this end, composing corresponding artifacts recursively by name and type, much like the mechanisms of hierarchy combination [14], mixin composition [15,16], superimposition [17,18], and higher-order hierarchies [19]. In contrast to these earlier approaches, for each artifact type a different implementation of the *composition operator* ('•') has to be provided in AHEAD (see. Sec. 2.3).

Figure 2 shows the features Base and Add containing source and non-source code artifacts. The feature equation '$\mathsf{Calc}_1 = \mathsf{Add} \bullet \mathsf{Base}$' composes both features. Feature composition is implemented as a (recursive) combination of the containment hierarchies of the features. For example, the resulting artifact Calc.java is composed of its counterparts in Base and in Add, matched by name and type.

2.3 Refinement of Software Artifacts

For each artifact type there is a distinct composition operator, i.e., a tool that composes the artifacts. We review a selection of tools available for several artifact types – most are integrated in the *AHEAD Tool Suite* [2].

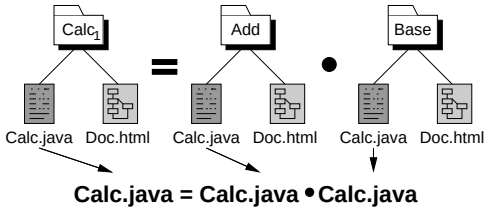

Fig. 2. Composing the containment hierarchies of two features.

Code Artifacts. Several languages and tools are available for composing source code artifacts [20,21,22,23,24,25,26]. For example, *Jak* [2] and *FeatureC++* [10] extend the syntax of Java and C++ by the keyword refines, which indicates that a class is refined. Figure 3 depicts a Java class Add (Lines 1–5) and a Jak refinement (Lines 6–11). The refinement adds a new field (Line 7) and extends an existing method via overriding (Lines 8–10). Calling Super invokes the refined method (Line 9). FeatureC++ refinements provide a similar syntax on top of C++ [10].

```
1  class Add {
2    int add(int a, int b) {
3      return a + b;
4    }
5  }
```

```
6  refines class Add {
7    int buf = 0;
8    int add(int a, int b) {
9      buf = Super.add(a, b); return buf;
10   }
11 }
```

Fig. 3. A class (top) and a refinement in Jak (bottom).

Non-Code Artifacts. Non-code artifacts can also be composed in an FOP fashion. For example, *Bali* is a tool for synthesizing program manipulation tools on the basis of extensible grammar specifications [27]. It defines grammars and refines them subsequently by composing grammar specifications.

Figure 4 depicts a simple grammar for processing arithmetic expressions (Lines 1–2) and a refinement that extends the rule Val for supporting real numbers (Line 3). During composition Bali grammars and their refinements are matched by their file names. The keyword Super refers to the refined grammar rule.

Another example is Xak. *Xak* is a language and tool for composing various kinds of XML documents [28]. It enhances XML by a module structure necessary for re-

```
1 Expr : Val | Val Oper Expr;
2 Val : INTEGER;
```

```
3 Val : Super.Val | REAL;
```

Fig. 4. A Bali grammar for processing arithmetic expressions (top) and a refinement for supporting real numbers (bottom).

finement. This way, a broad spectrum of software artifacts can be refined, since many artifacts can be expressed in XML, e.g., UML diagrams (XMI), build scripts (ANT), deployment descriptors, or XHTML. Figure 5 depicts an XHTML document embedded in a Xak module CalcDoc (Lines 1–10) as well as a refinement (Lines 11–17) that adds several visual elements by overriding the Xak tag Operations (Lines 13–16). The tag xak:extends denotes overriding and xak:super refers to the refined Xak tag (Line 14).

```
1 <html xmlns:xak="http://www.onekin.org/xak"
2   xak:artifact="CalcDoc" xak:type="xhtml">
3 <head><title>Calculator Documentation</title></head>
4 <body bgcolor="white">
5 <h1>Calculator Documentation</h1>
6   <ul xak:module="Operations">
7     <li>Addition of integers</li>
8   </ul>
9 </body>
10 </html>
```

```
11 <xak:refines xmlns:xak="http://www.onekin.org/xak"
12  xak:artifact="CalcDoc">
13   <xak:extends xak:module="Operations">
14     <xak:super xak:module="Operations"/>
15     <li>Subtraction of integers</li>
16   </xak:extends>
17 </xak:refines>
```

Fig. 5. A Xak/XHTML document (top) and a refinement (bottom).

Further types of code and non-code artifacts are, e.g., feature equations [2], feature models [29], grammar specifications [30], UML models [31,32], and aspects [6].

3 A Multi-Representation SPL

The *Calculator Software Product Line (CalcSPL)* is a multi-representation SPL that implements a simple calculator. It consists of several multi-representation features for

handling and processing arithmetic expressions and serves as our motivating example. Figure 6 depicts the feature diagram of CalcSPL[1]. CalcSPL supports basic arithmetic operations and provides command line access.

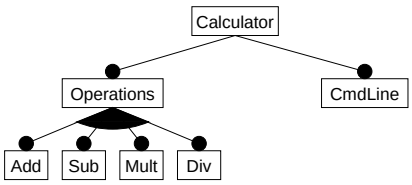

Fig. 6. Feature diagram of CalcSPL.

Figure 7 depicts CalcSPL's stack of features in top-down order as well as their internal artifacts. The remaining section describes the features in detail.

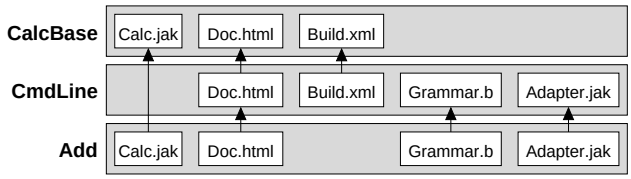

Fig. 7. Features of CalcSPL.

3.1 Feature CalcBase

`CalcBase` is the core of CalcSPL. It provides basic support for handling arithmetic expressions and is the basis of further features. `CalcBase` contains the following artifacts:

1. The Jak file `Calc.jak` contains a class `Calc` that provides the basic functionality for handling arithmetic expressions. The method `enter` stores an integer, `clear` initializes the calculator, and `top` returns the processed value:

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

[1] Boxes denote features and subfeatures; a filled bullet denotes a mandatory feature; a filled arc denotes a group of features in which at least one feature needs to be selected [33].

```
    e0 = e1 = e2 = 0;
  }
  String top() {
    return String.valueOf(e0);
  }
}
```

2. The HTML file `Doc.html` sets up the documentation of CalcSPL. Figure 8 depicts the documentation displayed in a browser; the subset introduced by `CalcBase` is annotated accordingly.

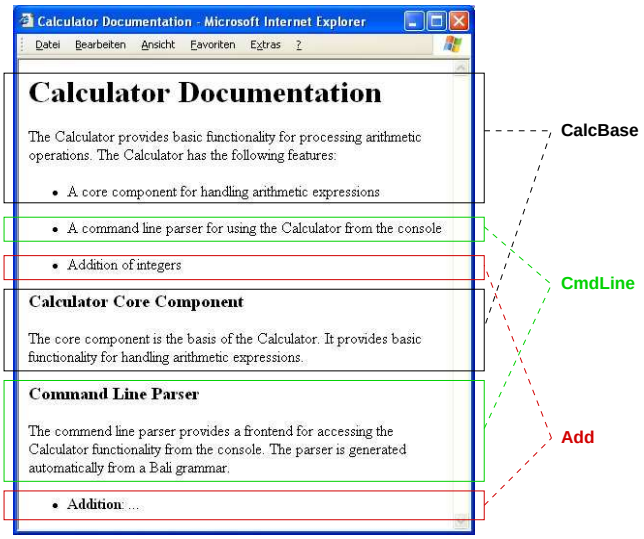

Fig. 8. HTML documentation of the CalcSPL.

3. The XML file `Build.xml` controls the build process of CalcSPL.

3.2 Feature CmdLine

`CmdLine` provides command line access to the calculator. It contains the following artifacts:

1. The Bali grammar `Grammar.b` is the basis for generating a parser that processes a user's input:

   ```
   Expr : Val | Val Oper Expr;
   Val  : INTEGER;
   ```

   It needs to be refined by concrete operations on integers.
2. A refinement of `Build.xml` adds commands for generating the parser from the grammar.

3. The class `Adapter` connects the parser and the class `Calc` to exchange information between the user and the calculator.
4. A refinement of `Doc.html` adds information about how to use the command line of the calculator (see Fig. 8).

3.3 Feature Add

The feature `Add` provides support to sum integers. It contains the following artifacts:

1. A refinement of the class `Calc` introduces a method `add` to sum two integers that have been entered into the calculator:

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

2. A refinement of the Bali grammar (`Grammar.b`) introduces the '+' operator, thus extending the parser to be generated:

```
"+" PLUS
Oper : super.Oper | PLUS;
```

3. A refinement of the class `Adapter` extends the information exchange between parser and the calculator.
4. A refinement of `Doc.html` lists 'Addition' as supported operation and explains the command line syntax for the addition of integers (see Fig. 8).

3.4 Further Features

The features `Sub`, `Mult`, and `Div` are implemented analogously to the feature `Add`.

4 The Role of *g*DEEP

As we have illustrated in the previous section, multi-representation features exist and need to be represented and composed. But the degree of detail in which contemporary models such as AHEAD describe what happens when refining and composing features is far too low. Every time we want to incorporate a new type of software artifact, we are forced to reinvent the wheel by developing a new composition tool. We need a formal model of feature composition and program refinement that is language-independent.

We propose such a formal model in form of a calculus. Our goal is twofold: (1) to provide deeper insight into the underlying principles of feature composition, and (2) to develop generic algorithms and tools that can be used to manipulate many kinds of software artifacts.

We call our calculus *g*DEEP, which is short for *generalized* DEEP. *g*DEEP generalizes the ideas of the DEEP calculus [4], which is a Java-like programming language and type system with built-in support for virtual classes and deep mixin composition [4].

Work on DEEP has demonstrated that it is possible to build a formal type theory that can handle feature composition.

The *g*DEEP calculus simplifies DEEP by stripping out all of the parts that are specific to Object-oriented Programming. What remains is a pure module system, and a set of rules for composing and refining modules.

4.1 The Module System

In *g*DEEP features are represented as modules, which have a deep hierarchical structure. A module may contain submodules, subsubmodules, and so forth. There are two kinds of modules with different composition semantics: (1) compound modules, and (2) atomic modules.

Compound modules have a named hierarchical substructure. Java packages and classes are good examples. Composing two compound modules together will compose submodules recursively with the same name and type.

Atomic modules do not have a particular substructure which *g*DEEP can intepret; an atomic module can be any arbitrary term in the artifact language. Examples are: Java methods and fields, XML code, Bali grammar rules, etc. When two atomic modules are composed, one simply overrides the other. However, the overriding definition may refer to the original definition using the original keyword (see Sec. 5), which is similar to the super keyword in Java. The original keyword allows the text of two atomic modules to be combined in a way that is specific to the particular modules and artifact language in question.

(Note that *g*DEEP uses original as a keyword, whereas AHEAD uses Super. We made this change to avoid confusion with the super keyword in Java.)

4.2 Feature Composition

In Figure 9, we illustrate the composition of compound and atomic modules. The modules *A* and *B* are compound modules, while *C*, *D*, and *E* are atomic. Modules are composed by deep mixin composition [15,4]. That is, *A* and *B* are composed recursively by composing their constituents. The refinement of submodule *D* uses the original keyword to refine the text of *D*.

4.3 Plugging Artifact Languages into *g*DEEP

When representing a software artifact in *g*DEEP, all of the structural elements in the artifact language must be mapped onto the two kinds of modules that are supported by *g*DEEP: compound modules and atomic modules.

Bali and XML are simple artifact languages, because they do not define any compound modules. As a result, they are very easy to plug into *g*DEEP. The *g*DEEP calculus provides a hierarchical module system, and all expressions in the artifact language become leaves (i.e. atomic modules) in the module hierarchy. The module calculus and the artifact language are almost completely orthogonal.

Java is a complex artifact language, because it defines its own compound modules in the form of packages, classes, and interfaces. Packages contain named subpackages

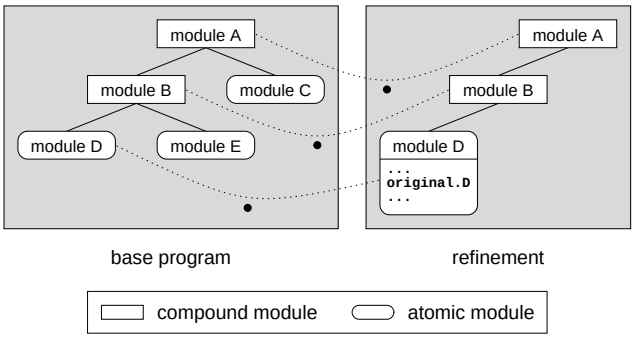

Fig. 9. Composition ('•') of compound and atomic modules.

and classes, while classes contain named inner classes, methods and fields. Methods and fields have no named substructure, so we represent them as atomic modules. We handle classes by defining a translation function that maps Java classes onto *g*DEEP compound modules. The translation function is one-to-one, so any manipulations performed within *g*DEEP can be mapped back to Java.

Once a software artifact has been translated to *g*DEEP, we can use the calculus to compose features in a language-independent way. Perhaps even more importantly, we can perform further analysis and manipulation steps, e.g. type-checking, consistency and error checking [34], or feature interaction analysis [35]. Figure 10 illustrates the role of *g*DEEP in software composition.

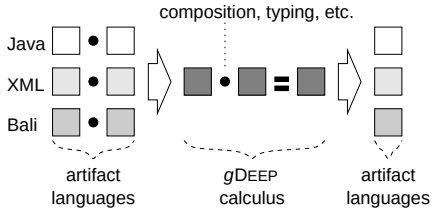

Fig. 10. Language-independent composition with *g*DEEP.

4.4 Pluggable Type Systems

In addition to providing an operational semantics for feature composition, the *g*DEEP calculus also defines a language-independent type system for features. This type system provides judgements that are needed to do modular type-checking, and which are not specific to any particular artifact language. It has two main capabilities.

First, there is a subtype relation defined over features and feature equations. Subtyping can be nominal, structural, or a mixture of both, and it supports multiple inheritance.

Features which require other features can express that dependency via subtyping. The type checker will ensure that all dependencies are properly declared as requirements, and that all requirements in a composition are satisfied.

Second, it is possible to plug artifact-specific type systems into the *g*DEEP type system. Current feature composition tools, such as AHEAD, do not support modular type checking. In the case of programs written in Java or other statically typed languages, all type checking must be done after composition. By plugging the artifact type system into *g*DEEP, it is possible to do type-checking before composition.

In the case of Java, the standard Java type system is responsible for assigning types to Java expressions, which always occur in atomic modules. The *g*DEEP type system is responsible for looking up class and method names within features, since names cross module boundaries. These two sets of judgements are largely orthogonal, so little work is needed to integrate the two.

An interesting interpretation of the role of *g*DEEP in type checking is the idea of *hierarchical type systems*. *g*DEEP is at the root of a hierarchy of type systems, because it provides basic capabilities that are common to all feature languages. Artifact-specific type systems extend *g*DEEP to check further artifact-specific properties.

4.5 *g*DEEP as a Formalism and a Tool

There are two ways of using *g*DEEP. Firstly, *g*DEEP is a means for formal reasoning about features and feature composition. It helps us to understand the general structure of features and the properties of feature composition, and it reveals the mandatory and optional properties that artifact languages must obey in order to support features and feature composition.

Secondly, *g*DEEP is an intermediate representation for language-independent feature composition. A concrete instantiation of *g*DEEP serves to express and reason about features in order to implement composition, consistency and error checking, feature interaction analysis and resolution, etc. [34,35].

Although, in Section 8, we outline an implementation of *g*DEEP and a tool that composes features based on our calculus, we focus here mainly on the mathematical formalism. Nevertheless, we believe that *g*DEEP will have an impact on both, the theory and practice of FOP.

5 Overview of the *g*DEEP Core Calculus

*g*DEEP by itself is not a full system. It is a base system into which another core calculus can be plugged, such as *Featherweight Java (FJ)* [36]. *g*DEEP provides a module system which supports feature composition. It does not handle the syntax, evaluation, or typing of concrete software artifacts; these must be provided by a "sister calculus" (e.g. FJ) that it is paired with. As examples, we will show how *g*DEEP can be combined with FJ and with proprietary formalizations of Bali and XML.

Figure 11 shows the syntax and Figure 12 shows the operational semantics of *g*DEEP. The syntax is divided into two parts. The first part, shown on the left, is a

Feature Calculus:

X, Y, Z	Variable (module)
ℓ	Slot labels (all)
$J, K, L \subset \ell$	Slot labels (module)
$j, k, l \subset \ell$	Slot labels (artifact)
M, N, O ::=	Terms:
X	variable
$\lambda X^+ \leq M.\ N$	function
$\mu X\{\overline{D}\}[\xi]$	module
$M(N)$	function application
$M@(N).L$	delegation
$M \otimes V$	composition
D, E ::=	Declarations:
$L : M$	nested module
$l : d$	artifact declaration

V, W ::=	Values:
$\lambda X^+ \leq M.\ N$	function
$\mu X\{\overline{D}\}[\xi]$	module

Artifact-specific constructs:

d, e ::=	Artifact-specific declarations:
...	(unspecified)
ξ ::=	Artifact-specific module info:
$\bullet$	empty
$\xi \otimes \xi$	composition
...	(unspecified)

Notation:

- $\overline{D}$ denotes a possibly empty sequence of declarations D_1 .. D_n, in which each declaration is terminated by a semicolon.
- D_L and D_l denote the declaration with label L or l, respectively, in the sequence $\overline{D}$.
- dom($\overline{D}$) denotes the set of labels in the sequence of declarations $\overline{D}$.
- $[X \mapsto M]\ N$ denotes the capture-avoiding substitution of term M for the variable X within N.

Syntax sugar:

- $M.L$ and $M.l$ are syntax sugar for $M@(M).L$ and $M@(M).l$, respectively.
- $\mathsf{original}_X.\ell$ (where ℓ can be L or l), is syntax sugar for $M@(X).\ell$, when the original keyword appears in the context:
  $M \otimes \mu X\{...\mathsf{original}_X.\ell...\}$. Furthermore, the X may be omitted, e.g. $\mathsf{original}.\ell$, in cases that are unambiguous.

Term Equality: α-renaming of bound variables x, plus

$$\frac{\overline{D} \text{ is a permutation of } \overline{E}}{\mu X\{\overline{D}\}[\xi]\ = \mu X\{\overline{E}\}[\xi]}$$

Fig. 11. Syntax of gDEEP

Evaluation Context: $E ::= [] \mid E(M) \mid V(E) \mid E@(M).L \mid V@(E).L \mid E \otimes V$

$$\frac{M \longrightarrow M'}{E[M] \longrightarrow E[M']} \quad \text{(E-CONG)}$$

$$\frac{L : N \in \overline{D}}{\mu X\{\overline{D}\}@(V).L \longrightarrow [X \mapsto V]N} \quad \text{(E-DLG)}$$

$$\frac{l : d \in \overline{D}}{\mu X\{\overline{D}\}@(V).l \longrightarrow [X \mapsto V]d} \quad \text{(EA-DLG)}$$

$$(\lambda X^{+} \leq M.\, N)(V) \longrightarrow [X \mapsto V]N \quad \text{(E-APP)}$$

$$\mu X\{\overline{C}\}[\xi_1] \otimes \mu X\{\overline{D}\}[\xi_2] \longrightarrow \mu X\{\overline{C} \uplus \overline{D}\}[\xi_1 \otimes \xi_2] \quad \text{(E-COMP)}$$

$$\text{where } \overline{C} \uplus \overline{D} = \overline{E} \text{ such that } \mathrm{dom}(\overline{E}) = \mathrm{dom}(\overline{C}) \cup \mathrm{dom}(\overline{D})$$
$$\text{and } E_\ell = \begin{cases} D_\ell & \text{if } \ell \in \mathrm{dom}(\overline{D}) \\ C_\ell & \text{otherwise} \end{cases}$$

$$\bullet \otimes \bullet = \bullet \quad (\xi\text{-COMP})$$

Fig. 12. Operational semantics of gDEEP

calculus for features. This part represents the core calculus proper, and it is the only part which is common across all applications of gDEEP.

The second part of the calculus, shown on the right, is *artifact-specific*. These terms serve as placeholders for the particular language to which gDEEP is being applied – the *target artifact language*, or *target language*. When support for features is added to Java, these terms are "filled in" with Java constructs. When support for features is added to XML, they are XML trees, and so on. The precise rules by which a target artifact language can be integrated with gDEEP will be described in later sections.

5.1 Records

In gDEEP, the classes and modules of a base program and features are encoded as recursive records. Records are declared using the syntax $\mu X\{\overline{D}\}[\xi]$, where $\overline{D}$ is a sequence of declarations. ξ is a artifact-specific annotation of some kind. It is used by the artifact language for the composition of artifact-specific details, but is otherwise ignored by the core calculus. (see Sec. 5.5).

The variable X provides a name for "self" within the record, much like the keyword this in C++ or Java. Records in gDEEP can be nested, so it is important that each record has a unique name for "self". The self-variable allows any declaration within a record to refer to other declarations within the same record by means of a *path*. The semantics of paths are somewhat subtle, and are explained in Section 5.2.

Declarations. There are two basic kinds of declaration:

1. $L : M$ declares a nested compound module.
2. $l : d$ declares a nested atomic module. The term d is an arbitrary declaration in the artifact language.

Atomic modules are treated as raw "chunks of code" and are not otherwise interpreted by *g*DEEP.

Composition. Feature composition is represented by recursive module composition. Module composition in *g*DEEP is very similar to object-oriented inheritance. The equation $N = M \otimes \mu X\{...\}$ is analogous to the Java syntax:

```
class N extends M { ... }
```

The expression $M \otimes V$ combines the declarations in M and the declarations in V to a single record. Conflicts are resolved by overriding; if M and V both have a declaration with the same label, then the one in V overrides the one in M. In order to be type-safe, declarations in V which are overriding must be subtypes of those in M (see Sec. 5.4).

5.2 Paths and Delegation

Delegation, written using the syntax $M@(N).L$, projects the declaration named L from the record M. Any occurrences of the self-variable X will be bound to N. Declarations in *g*DEEP are similar to methods in object-oriented languages. Each record comes equipped with a self-variable, and the body of a declaration may refer to "self" using that variable. In most object-oriented languages, including C++ and Java, "self" is treated as a hidden argument that is passed implicitly during a method call. In *g*DEEP, the "self" argument is not hidden, but is passed explicitly.

Usually, M and N are the same term, in which case the standard object-oriented dot notation $M.L$ can be used as syntax sugar for $M@(M).L$. The expression $M.L$ projects the slot named L from M, passing M as "self." The case where M and N are different arises when a refinement wishes to delegate behavior to a base program, as is discussed later.

The following example demonstrates how delegation works in the simple case: expressions of the form $M.L$. For clarity, we use Java as the artifact language:

```
let M1 ≝ μX{
  A :: μY{ int a; };
  B :: X.A ⊗ μZ{ int b; };
}
```

In this example, M_1 is defined as a record which contains the two nested records A and B. A is stand-alone definition, while B inherits from A. Note that B refers to A using the self-variable X.

Like object-oriented languages, *g*DEEP uses *late binding*. The variable X is not assigned a value until B is actually projected from M_1, e.g.,

$$M_1.B \longrightarrow M_1.A \otimes \mu Z\{\mathsf{int}\ b;\} \longrightarrow\!\!\!\!\!\rightarrow \mu Z\{\mathsf{int}\ a;\ \mathsf{int}\ b;\}$$

In the next example, we create a new record M_2, which extends M_1 by adding a new field named a_2 to A:

```
let M2 ≝ M1 ⊗ μX{
  A :: original_X.A ⊗ μY{ int a2; };
}
```

Because of late binding, extending the definition of A will automatically extend the definition of B. M_2 inherits B from M_1. However, when the expression $M_2.B$ is evaluated, X will be bound to M_2 rather than M_1, e.g.,

$$M_2.B \longrightarrow M_2.A \otimes \mu Z\{\mathsf{int}\ b;\} \longrightarrow\!\!\!\!\rightarrow \mu Z\{\mathsf{int}\ a;\ \mathsf{int}\ a2;\ \mathsf{int}\ b;\}$$

The definition of M_2 given above also introduces a more complex use of delegation, which is similar to further binding in BETA. Notice that the record A is overridden with a version that inherits from $\mathsf{original}_X.A$. The "original" module in this case is M_1, so $\mathsf{original}_X.A$ is syntax sugar for $M_1@(X).A$. The expression $M_1@(X).A$ means: "extract the declaration named A from M_1, but pass X, which is the self-variable for M_2, as self."

When a derived module inherits from a base module, it may wish to delegate some behavior to its parent. This has shown to be useful in program refinement via features [2,9]. Java provides a special keyword for delegation, called super, and the syntax given here mimics that behavior more formally. Note that we cannot use simple dot notation to perform delegation. The expression $M_1.A$ projects A from M_1, but it passes M_1 as self, which defeats the whole point of late binding. In order to inherit from A properly, we need to project A from M_1, using the self variable for M_2.

Type Constraints. It is neither sensible, nor type-safe, to pass just any term as self. A declaration $M@(N).L$ has a type constraint; it is only well-formed if N is subtype of M. In the case of the simple dot notation, $M.L$, this constraint is trivially satisfied. It is also safe for a derived module to delegate to a base module.

5.3 Extensible Fixpoints

The mechanism for late binding described above is one of the key technical innovations that gDEEP inherits from DEEP. As the syntax suggests, a record $\mu X\{...\}$ denotes a fixpoint. The formal theory of fixpoints is well-established, but most formal calculi expand the fixpoint immediately at the point of declaration. This immediate expansion prevents a recursive structure from being extended.

gDEEP differs from other calculi because it uses *extensible fixpoints*. A self-variable is not bound at the point of declaration, it is bound at the point of use. This is a prerequisite for modeling feature composition based on recursive module composition.

5.4 Monotone Functions

In gDEEP, functions are used to represent features. Like many other module calculi, gDEEP allows functions to be defined over modules. The expression $\lambda X^+ \leq M.\ N$ is

a function that accepts any subtype of M as an argument. The expression $Add(Base)$ will call a function Add with the argument $Base$ (cf. Sec. 2). As is standard practice, functions which accept multiple arguments are encoded by currying.

Similar to generic Java and System $F_{\leq}$, *g*DEEP uses *bounded quantification* [37] to establish type constraints on formal arguments. Bounded quantification relies on the subtype relation, rather than signatures or kinding.

Unlike most other module calculi, functions in *g*DEEP are *monotone*, which is why they are declared with the curious X^+ notation on the variables. If F is a monotone function and A and B are base programs, then:

$$\forall A, B.\ A \leq B \text{ implies } F(A) \leq F(B)$$

This property is enforced in *g*DEEP because functions are primarily used to encode features. It is certainly possible to include general-purpose (i.e. non-monotone) functions as well, but such flexibility is overkill for modeling features, and would needlessly complicate the calculus.

A feature which is applicable to a base program A can be declared using the following syntax:

$$\lambda X^+ \leq A.\ X \otimes \mu Y\{...\}$$

In other words, a feature takes an argument X, and extends it by adding the declarations given in $\{...\}$. If we were to write out this definition using a Java-like syntax, it would look like:

```
class F <X extends A> extends X { ... }
```

Note that this declaration is illegal in Java, because Java does not allow a class to inherit from a type argument. *g*DEEP supports features by removing this restriction, and establishing a monotonicity constraint so that feature compositions obey sensible subtyping laws.

Subtyping Laws for Features. A feature encapsulates a slice of program behavior. Applying a feature to a program will extended the functionality of the program in some way. This leads us to two typing laws for features. If F is a feature and A and B are base programs, then:

$$\forall A.\ F(A) \leq A$$
$$\forall A, B.\ A \leq B \text{ implies } F(A) \leq F(B)$$

The first law states that a feature always extends its argument. We can express the first law as a subtyping rule by saying that F is a feature if and only if $F \leq \lambda X^+.\ X$. The second law (monotonicity) states that if a feature is applied to a more specific program, it will always generate a more specific result.

Together, these two laws allow us to derive subtyping rules for feature compositions. If $F_1..F_n$ are a set of features, and $G_1..G_m$ are a set of features, we can say that

$$F_1(F_2(...F_n(A))) \ \leq \ G_1(G_2(...G_m(A)))$$

if and only if $F_1..F_n$ contains all of the features in $G_1..G_n$, *and all of those features are applied in the same order*.

This rule is somewhat similar to multiple inheritance, but it includes a restriction on the order in which features are applied. Two expressions which apply the same features, but apply them in a different order, e.g. $F(G(A))$ and $G(F(A))$, will generate programs with different behavior, as common in FOP. The subtype rules given here neatly capture this distinction.

5.5 Artifact-Specific Constructs

gDEEP provides two main "hooks" for integration with a particular artifact language. Declarations in the artifact language are denoted by d. In the case of Java, d represents field and method declarations. Artifact declarations are completely opaque; gDEEP does not interpret them in any way.

However, there are some cases where an expression in the target language should not be opaque. For example, a Java class is somewhat similar to a record in gDEEP. A class contains a set of named fields and methods, and we would like to use feature composition to extend classes.

gDEEP makes classes transparent by modeling them as a special kind of record. Records can be annotated with a domain-specific construct, named ξ. ξ is defined to hold all of the specific information about classes that Java requires, but that gDEEP does not include. In our encoding of FJ, we use it to hold the class constructor, along with the extends clause. Using this information, is is possible to create a one-to-one mapping between Java classes and records in gDEEP, as we will show in Section 6.

5.6 Type System

The type system of gDEEP is based on the type system of DEEP. A full discussion of meta-theory and a proof of type safety is well beyond the scope of this paper; please see the original paper on DEEP [4] for details. Figures 13 and 14 depict the type rules of gDEEP.

The type system defines three relations over terms: $\leq$, $\equiv$, and $\lessdot$. For compactness, we use $\lhd$ as a meta-variable which ranges over all three relations.

The judgement $M \leq N$ means that M is a subtype of N. gDEEP supports both structural and nominal subtyping. Structural subtyping compares records and functions by comparing their slots and bodies. Nominal subtyping compares names in the form of variables and paths, e.g., $X.A \leq X.A$.

The judgement $M \equiv N$ means that M is equivalent to N. Two terms are equivalent if they reduce to a common result.

The judgement $M \lessdot N$ means that N is a minimal upper bound of M. This relation is very similar to subtyping, except that it does not discard information. The statement $M \lessdot R$, where R is a record, says that R is the most specific interface that can be derived for M.

These three relations do double-duty. First, they can be used to compare two terms to see whether one is related to another; e.g., given M and N, we wish to know whether

$\Gamma ::=$	Contexts	$\lhd ::=$	Subtype judgements
$\emptyset$	empty context	$\leq$	subtype
$\Gamma, X \leq M$	variable bound	$\lessdot$	minimal subtype
...	*artifact specific*	$\equiv$	term equivalence

Notation:

- $\Gamma \vdash J_1, J_2$ is shorthand for $\Gamma \vdash J_1$ and $\Gamma \vdash J_2$.
- $\mathrm{fv}(M)$ denotes the set of free variables in M.
- $\mathrm{iv}(M)$ denotes the set of free variables that occur in invariant positions in M.
  A variable occurs in an invariant position if it occurs within I in the following terms: $\lambda X^+ \leq I.\ M$, or $M@(I).L$.

Subtyping: $\boxed{M \lhd M, D \lhd D}$

$$\frac{\Gamma \vdash\ M \equiv N}{\Gamma \vdash\ M \lessdot N} \qquad \frac{\Gamma \vdash\ M \lessdot N}{\Gamma \vdash\ M \leq N} \tag{S-Weak1-2}$$

$$\frac{\Gamma \vdash\ M \lhd N, \quad N \lhd O, \quad N \text{ wf}}{\Gamma \vdash\ M \lhd O} \tag{S-Trans}$$

$$\frac{\Gamma \vdash\ N \equiv M}{\Gamma \vdash\ M \equiv N} \tag{S-Sym}$$

$$\frac{M \longrightarrow N}{\Gamma \vdash\ M \equiv N} \tag{S-Red}$$

$$\Gamma \vdash\ X \equiv X \qquad \frac{X \leq M \in \Gamma}{\Gamma \vdash\ X \lessdot M} \tag{S-Var / SE-Var}$$

$$\frac{\Gamma \vdash\ M \lhd M', \quad N \equiv N'}{\Gamma \vdash\ M(N) \lhd M'(N')} \qquad \frac{\Gamma \vdash\ N \leq N'}{\Gamma \vdash\ M(N) \leq M(N')} \tag{S-App1 / S-App2}$$

$$\frac{\Gamma \vdash\ M \lhd M', \quad N \equiv N'}{\Gamma \vdash\ M@(N).L \lhd M'@(N').L} \tag{S-Dlg}$$

$$\frac{\Gamma \vdash\ M \lhd M', \quad V \lhd V'}{\Gamma \vdash\ M \otimes V \lhd M' \otimes V'} \qquad \Gamma \vdash\ M \otimes V \leq M \tag{S-Cmp / S-Sup}$$

$$\frac{\begin{array}{c}\Gamma \vdash\ M \equiv M' \\ \Gamma, X \leq M \vdash\ N \lhd N'\end{array}}{\Gamma \vdash\ \lambda X^+ \leq M.\ N \lhd \lambda X^+ \leq M'.\ N'} \tag{S-Fun}$$

$$\frac{\begin{array}{c}\mathrm{dom}(\overline{D})\ \{\lhd\}\ \mathrm{dom}(\overline{E}) \\ \Gamma, X \leq \mu X\{\overline{D}\}[\xi_1] \vdash\ \xi_1 \lhd \xi_2, \quad \forall \ell \in \mathrm{dom}(\overline{E}).\ D_\ell \leq E_\ell\end{array}}{\Gamma \vdash\ \mu X\{\overline{D}\}[\xi_1] \lhd \mu X\{\overline{E}\}[\xi_2]} \tag{S-Rec}$$

where
$S\ \{\leq\}\ S'$ iff $S \supseteq S'$
$S\ \{\lessdot\}\ S'$ iff $S = S'$
$S\ \{\equiv\}\ S'$ iff $S = S'$

$$\frac{\Gamma \vdash\ M \lhd N}{\Gamma \vdash\ L : M \lhd L : N} \tag{S-Dcl}$$

$$\frac{\textit{artifact specific}}{\Gamma \vdash\ l : d \lhd l : e} \qquad \frac{\textit{artifact specific}}{\Gamma \vdash\ \xi_1 \lhd \xi_2} \tag{S-Art1-2}$$

Fig. 13. Subtyping rules of gDEEP.

Well-formedness: $\boxed{\Gamma \text{ wf}, M \text{ wf}, D \text{ wf}}$

$$\emptyset \text{ wf} \qquad \frac{\Gamma \text{ wf} \quad \Gamma \vdash M \text{ wf} \quad X \notin \text{dom}(\Gamma)}{\Gamma, X \leq M \text{ wf}} \tag{W-CTX1-2}$$

$$\frac{\begin{array}{l}\Gamma \vdash M \text{ wf}, \quad M \lessdot \lambda X^{+} \leq N'.\, O \\ \Gamma \vdash N \text{ wf}, \quad N \leq N' \\ \Gamma \vdash [X \mapsto N]\, O \text{ wf}\end{array}}{\Gamma \vdash M(N) \text{ wf}} \tag{W-APP}$$

$$\frac{\begin{array}{l}\Gamma \vdash M \text{ wf}, \quad M \leq \mu X\{\overline{D}\}[\xi] \quad \ell \in \text{dom}(\overline{D}) \\ \Gamma \vdash N \text{ wf}, \quad N \leq M\end{array}}{\Gamma \vdash M@(N).\ell \text{ wf}} \tag{W-DLG}$$

$$\frac{\Gamma \text{ wf}, \ X \leq M \in \Gamma}{\Gamma \vdash X \text{ wf}} \tag{W-VAR}$$

$$\frac{X \notin \text{iv}(N) \qquad \Gamma \vdash M \text{ wf} \quad \Gamma, X \leq M \vdash N \text{ wf}}{\Gamma \vdash \lambda X^{+} \leq M.\, N \text{ wf}} \tag{W-FUN}$$

$$\frac{\Gamma, X \leq \mu X\{\overline{D}\}[\xi] \vdash \overline{D} \text{ wf}, \ \xi \text{ wf}}{\Gamma \vdash \mu X\{\overline{D}\}[\xi] \text{ wf}} \tag{W-REC}$$

$$\frac{\begin{array}{l}\Gamma \vdash M \text{ wf}, \quad M \lessdot \mu X\{\overline{C}\}[\xi_1] \qquad \mathcal{S} = \text{dom}(\overline{C}) \cap \text{dom}(\overline{D}) \\ \Gamma, X \leq M \otimes \mu X\{\overline{D}\}[\xi_2] \vdash \xi_1 \otimes \xi_2 \text{ wf}, \quad \overline{C} \uplus \overline{D} \text{ wf}, \quad \forall \ell \in \mathcal{S}.\, D_\ell \leq C_\ell\end{array}}{\Gamma \vdash M \otimes \mu X\{\overline{D}\}[\xi_2] \text{ wf}} \tag{W-CMP}$$

$$\frac{\Gamma \vdash M \text{ wf}}{\Gamma \vdash L : M \text{ wf}} \qquad \frac{\textit{artifact specific}}{\Gamma \vdash l : d \text{ wf}} \tag{W-DCL1-2}$$

Fig. 14. Wellformedness rules of gDEEP.

$M \leq N$. Second, given a single term M, we wish to find a bounding type for M, a record or function V such that $M \leq V$.

These two uses require different algorithms; the algorithm for finding a supertype is different from the one for comparing two terms. Logically, however, both algorithms implement the same mathematical relations. This property is easiest to see for $\equiv$. The algorithm for finding a V such that $M \equiv V$ is the same as evaluation; we evaluate M to obtain V. Supertypes and minimal supertypes can be computed in an analagous manner.

Type Checking. Since the subtype relation can be used to derive upper bounds, there is no need for a conventional typing relation. Instead, type-checking is done by ensuring that terms are well-formed. The well-formedness judgement does the following checks:

- It ensures that every function (i.e., every feature) is monotone.
- For a function call $F(A)$, it ensures that F is a function and A has the correct type.
- For delegation $M@(N).L$, it ensures that $N \leq M$, and M is a record with a slot named L.
- For $M \otimes V$, it ensures that every overriding slot in V is a subtype of the corresponding slot in M.

Type-checking is modular, unlike existing tools for FOP [2,10]. Each feature is type-checked before it is composed. Unfortunately, not all type errors can be caught before composition, as shown in the following example, which uses Java as a target artifact language.

```
A = μX { String a; };
B = A ⊗ μX{
  String foo(int i) { return a + i; }
};
F = λX+ ≤ A. X ⊗ μY{
  int foo(int i) { return i + a; }
};
C = F(B)
```

In this example, A and B have no type errors. The feature F introduces one type error, because it tries to add a string to an integer, which is not allowed in Java. This error will be caught immediately. The declaration of C introduces a second error, because the type signature for foo provided by F conflicts with the one already present in B.

Java 1.5 avoids the second error by making the declaration of F illegal — it is not possible to inherit from a variable. gDEEP removes this restriction (thus, supporting feature composition), but it must do type-checking in two passes. The body of F is checked once at the point where F is declared, and a second time when F is applied to B. The second check will catch naming and type conflicts that are specific to a particular composition.

x, y, z	Variables
$m, n \subset l$	method names
$f, g \subset l$	field names
$b, c \subset L$	class names
A, B, C ::=	Class
Object	universal supertype
M	gDeep term
CL ::=	Class declaration
class c extends C $\{\overline{FD}\ KD\ \overline{MD}\}$	
FD ::= $C\ f$;	Field declaration
KD ::=	Constructor declaration
$c(\overline{C}\ \overline{x})$ {super($\overline{t}$); this.$\overline{f} = \overline{t}$;}	
refines $c(\overline{C}\ \overline{x})$ {this.$\overline{f} = \overline{t}$;}	
MD ::=	Method declaration
$C\ m(\overline{C}\ \overline{x})$ {return t;}	

d, e ::= FD | MD	Declarations
s, t, u ::=	Terms
x	variable
instance $C\ \{\overline{f} = \overline{t}\}$	(object)
$t.f$	field access
$t.m(\overline{u})$	method access
$C@(t).m(\overline{u})$	delegation
new $C(\overline{t})$	constructor call
$(C)\ t$	cast
v, w ::=	Values
instance $C\ \{\overline{f} = \overline{v}\}$	object
ξ ::=	Extra class info
(extends C, KD)	superclass + constructor
$\xi \otimes \xi$	composition

Mapping from *g*FJ to *g*DEEP:

$$\langle\!\langle \text{class } c \text{ extends } B\ \{\overline{FD}\ KD\ \overline{MD}\} \rangle\!\rangle =$$
$$\mu X\{\langle\!\langle \overline{FD} \rangle\!\rangle\ \langle\!\langle \overline{MD} \rangle\!\rangle\}\ [(\text{extends } B, KD)] \text{ where } X \text{ chosen fresh}, \quad KD = c(\overline{B}\ \overline{x})\{...\}$$
$$\langle\!\langle C\ f \rangle\!\rangle = f : C\ f$$
$$\langle\!\langle C\ m(\overline{B}\ \overline{x})\ \{\ \text{return } t;\} \rangle\!\rangle = m : C\ m(\overline{B}\ \overline{x})\ \{\ \text{return } t;\}$$

$$(\text{extends B},\ c(\overline{C}\ \overline{x})\ \{\text{super}(\overline{s});\ \text{this}.\overline{f} = \overline{t};\}) \otimes (\text{extends B},\ \text{refines } c(\overline{C}\ \overline{x})\ \{\text{this}.\overline{g} = \overline{u};\}) =$$
$$(\text{extends B},\ c(\overline{C}\ \overline{x})\ \{\text{super}(\overline{s});\ \text{this}.\overline{f} = \overline{t};\ \text{this}.\overline{g} = \overline{u};\})$$

Syntax Sugar: original.$m(\overline{t})$ is syntax sugar for $C@(\text{this}).m(\overline{t})$, when it appears within the context $C \otimes \langle\!\langle \text{class } c \text{ extends } B\ \{...\text{original}.m(\overline{t})...\}$

Fig. 15. Syntax and translation of *g*FJ

Evaluation Context:

$$E ::= [] \mid E.f \mid E.m(\overline{t}) \mid v.m(\overline{w}, E, \overline{t}) \mid C@(E).m(\overline{t}) \mid C@(v).m(\overline{w}, E, \overline{t})$$
$$\mid \mathsf{new}\ C(\overline{v}, E, \overline{t}) \mid (C)E \mid \mathsf{instance}\ \ C\ \{\overline{f} = \overline{v},\ g = E,\ \overline{f'} = \overline{t}\}$$

Reduction: $\boxed{t \longrightarrow t}$

$$\frac{t \longrightarrow t'}{E[t] \longrightarrow E[t']} \qquad \text{(E-CONGRUENCE)}$$

$$\mathsf{instance}\ \ C\ \{\overline{f} = \overline{v}\}.f_i \longrightarrow v_i \qquad \text{(E-FIELD)}$$

$$\frac{v = \mathsf{instance}\ \ C\ \{\overline{f} = \overline{u}\} \quad \text{method}(C, m) = B\ m(\overline{A}\ \overline{x})\ \{\mathsf{return}\ \ t;\}}{v.m(\overline{w}) \longrightarrow [\mathsf{this} \mapsto v][\overline{x} \mapsto \overline{w}]t} \qquad \text{(E-METH)}$$

$$\frac{v = \mathsf{instance}\ \ D\ \{\overline{f} = \overline{u}\} \quad \text{method}(C, m) = B\ m(\overline{A}\ \overline{x})\ \{\mathsf{return}\ \ t;\}}{C@(v).m(\overline{w}) \longrightarrow [\mathsf{this} \mapsto v][\overline{x} \mapsto \overline{w}]t} \qquad \text{(E-DLG)}$$

$$\frac{\text{construct}(C, \overline{v}) \stackrel{\text{def}}{=} \overline{f}\ \overline{t}}{\mathsf{new}\ C(\overline{v}) \longrightarrow \mathsf{instance}\ C\ \{\overline{f} = \overline{t}\}} \qquad \text{(E-NEW)}$$

$$\frac{\phi \vdash\ C <: B}{(B)\ \mathsf{instance}\ C\{\overline{f} = \overline{t}\} \longrightarrow \mathsf{instance}\ C\{\overline{f} = \overline{t}\}} \qquad \text{(E-CAST)}$$

Java Subtyping: $\boxed{C <: C}$

$$\Gamma \vdash\ C <: \mathsf{Object} \qquad \frac{\Gamma \vdash\ C \equiv B}{\Gamma \vdash\ C <: B} \qquad \frac{\Gamma \vdash\ A <: B \quad B <: C}{\Gamma \vdash\ A <: C}$$

$$\frac{\Gamma \vdash\ C \leq \langle\!\langle \mathsf{class}\ c\ \mathsf{extends}\ B\ \{\overline{FD}\ KD\ \overline{MD}\}\rangle\!\rangle}{\Gamma \vdash\ C <: B}$$

Method and Constructor Lookup:

$$\frac{C \longrightarrow\!\!\!\!\!\rightarrow \langle\!\langle \mathsf{class}\ c\ \mathsf{extends}\ B\ \{\overline{FD}\ KD\ \overline{MD}\}\rangle\!\rangle}{\text{method}(C, m) \stackrel{\text{def}}{=} MD_m} \qquad \frac{C \longrightarrow\!\!\!\!\!\rightarrow \langle\!\langle \mathsf{class}\ c\ \mathsf{extends}\ B\ \{\overline{FD}\ KD\ \overline{MD}\}\rangle\!\rangle \quad m \notin \text{dom}(\overline{MD})}{\text{method}(C, m) \stackrel{\text{def}}{=} \text{method}(B, m)}$$

$$\text{construct}(\mathsf{Object}, \bullet) \stackrel{\text{def}}{=} \bullet \qquad \frac{\begin{array}{c} C \longrightarrow\!\!\!\!\!\rightarrow \langle\!\langle \mathsf{class}\ c\ \mathsf{extends}\ B\ \{\overline{FD}\ KD\ \overline{MD}\}\rangle\!\rangle \\ KD = c(\overline{C'}\ \overline{x})\ \{\mathsf{super}(\overline{t});\ \mathsf{this}.\overline{g} = \overline{u};\} \end{array}}{\text{construct}(C, \overline{s}) \stackrel{\text{def}}{=} \text{construct}(B, [\overline{x} \mapsto \overline{s}]\overline{t}),\ \overline{g}\ [\overline{x} \mapsto \overline{s}]\overline{u}}$$

Fig. 16. Operational semantics of *g*FJ

6 Generalized Featherweight Java

Featherweight Java (FJ) is a minimal core calculus for Java, which models the type system and the core semantics of the Java language [36]. We have developed *generalized Featherweight Java (g*FJ*)* on top of FJ, so that we can translate and plug it into *g*DEEP. Figure 15 depicts the syntax and transformation rules and Figure 16 depicts operational semantics of *g*FJ.

6.1 Classes and Delegation

*g*FJ extends FJ in two fundamental ways. First and foremost, it eliminates the global class table. FJ uses a flat class table, in which class names c are mapped to class declarations CL. In contrast, *g*FJ refers to classes using using arbitrary *g*DEEP expressions, rather than simple names. In most cases, a *g*FJ program will refer to a class c using a local path $X.c$, where X is the self-variable for the enclosing module.

Second, *g*FJ introduces a syntax for delegating behavior from the refinement of a method to the original definition of that method. This mechanism is essentially the same as module delegation in *g*DEEP, and is much like the super keyword in Java.

When evaluating the term $C@(t).m(\overline{u})$, *g*DEEP will look up the method m in class C, and then call that method with $\overline{u}$ as arguments, passing t as the target object. In practical programming, t is always this, and C is the original version of the class currently being refined, so we introduce some syntax sugar: original.$m(\overline{u})$. (The original keyword replaces the Super keyword which is used in AHEAD).

6.2 Constructors and Instances

Our goal in formalizing *g*FJ was to remain as close to the definition of FJ as possible. We replace class names with arbitrary *g*DEEP expressions, and add delegation, since these concepts are fundamental to FOP. However, we would like to avoid any other changes to the core FJ syntax.

Unfortunately, FJ has one significant limitation that requires us to make further changes. The actual extension is not as interesting as the fact that we were forced to make this modification in the first place. Although support for features can be added to different languages, they cannot be added to just *any* language. Some languages, including FJ, are not "feature-friendly" or "feature-ready". Although this section deals with FJ in particular, it is a case study in the kinds of constraints that FOP places on language design.

In Java, the term new $C(\overline{t})$ is an expression, which creates an instance of class C by evaluating the constructor for C. FJ deliberately simplifies this mechanism in an effort to make the core calculus as small as possible. In FJ, the term new $C(\overline{t})$ is not an expression. It is a value which represents an instance of class C.

The problem with using new $C(\overline{t})$ as a value is that every field in the class must be listed as an argument in the constructor. This limitation means that a feature cannot add a new field to a class without changing the constructor signature, and changing the signature will break any code which uses the class. In a feature-friendly language,

constructors should be "virtual". It should be possible to refine a constructor without altering its signature, much like a overriding a virtual method.

In order to support constructor refinement, gFJ follows the Java model, rather than the FJ model. The expression new $C(\bar{t})$ will evaluate the constructor for C, and return an instance, which is represented by the syntax instance $C\{\overline{f} = \bar{t}\}$. All fields must be initialized within the constructor, but they do not have to appear as constructor arguments.

6.3 Type System

Figures 17 and 18 depict the type rules of gFJ. We did not make any changes to inheritance, although we could have. Feature composition is very similar to inheritance, and gDEEP already provides a subtype relation. Thus, we could have drafted a smaller and more elegant calculus by using gDEEP to handle inheritance, subtyping, and method lookup.

There are two reasons why we did not attempt such a simplification. First, it would change the type system dramatically, so the resulting calculus would bear little resemblance to either FJ, or full Java. Second, there is a subtle semantic difference between inheritance and feature refinement. With inheritance, a derived class can have different constructors than the base class. Feature refinement is more strict; constructors must keep the same signature.

6.4 Translation

The core of the encoding of gFJ is the translation between Java-like syntax and gDEEP syntax. The translation function, denoted by $\langle\!\langle \_ \rangle\!\rangle$, provides a one-to-one mapping between gFJ declarations and gDEEP declarations (cf. Fig. 15). Because the function is bijective, it is also possible to convert gDEEP terms back to gFJ.

Classes in gFJ map to records in gDEEP, while fields and methods map to atomic declarations within those records. (gFJ splits the set l of slot labels into subsets m and f, containing field names and method names, respectively.) The superclass and constructor are placed in the artifact-specific term $[\xi]$. This encoding demonstrates how ξ is intended to be used; it is a mechanism for capturing artifact-specific information that does not fit into standard gDEEP records.

gFJ also provides an artifact-specific way of composing constructors and extends clauses. Two classes can only be composed if they have the same superclass, and if the constructor on the right refines the constructor on the left. The constructor for a class is responsible for initializing all of its fields. Composing two constructors together will merge the two sets of initialization statements. The refines keyword ensures that the two definitions do not call the superclass constructor in different ways.

6.5 Formalities

The operational semantics of gFJ is largely self-contained, and fairly similar to FJ. However, gFJ relies on gDEEP for one thing: every class expression C must be reduced to a class declaration CL. The helper functions "method" and "construct" contain

Syntax:

$$\begin{array}{rl} \Gamma ::= & \ldots \\ & \Gamma, x : T \text{ variable type} \end{array}$$

Typing: $\boxed{t : C}$

$$\frac{x : C \in \Gamma}{\Gamma \vdash\ x : C} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash\ \text{fields}(C) = \overline{f}\ \overline{B}, \quad \overline{t} : \overline{B}}{\Gamma \vdash\ \mathsf{instance}\ C\{\overline{f} = \overline{t}\} : C} \quad \text{(T-OBJECT)}$$

$$\frac{\Gamma \vdash\ t : B, \quad \text{fields}(B) = \overline{f}\ \overline{C}}{\Gamma \vdash\ t.f_i : C_i} \quad \text{(T-FIELD)}$$

$$\frac{\Gamma \vdash\ t : C, \quad C@(t).m(\overline{u}) : B}{\Gamma \vdash\ t.m(\overline{u}) : B} \quad \text{(T-METH)}$$

$$\frac{\begin{array}{c} \Gamma \vdash\ t : C', \quad C' \leq C \\ \Gamma \vdash\ \text{mtype}(C, m) = \overline{A} \to B, \quad \overline{u} : \overline{A} \end{array}}{\Gamma \vdash\ C@(t).m(\overline{u}) : B} \quad \text{(T-DLG)}$$

$$\frac{\begin{array}{c} \Gamma \vdash\ \overline{t} : \overline{A} \\ \Gamma \vdash\ C \leq \langle\!\langle \mathsf{class}\ c\ \mathsf{extends}\ B\{\overline{FD}\ KD\ \overline{MD}\}\rangle\!\rangle \\ KD = c(\overline{A}\ \overline{x})\ \{\mathsf{super}(\overline{t});\ \mathsf{this}.\overline{g} = \overline{u};\} \end{array}}{\Gamma \vdash\ \mathsf{new}\ C(\overline{t}) : C} \quad \text{(T-NEW)}$$

$$\frac{\Gamma \vdash\ t : B, \quad C <: B}{\Gamma \vdash\ (C)\, t : C} \quad \text{(T-CAST)}$$

$$\frac{\Gamma \vdash\ t : B, \quad C \not<: B, \quad B \not<: C \qquad \textit{stupid warning}}{\Gamma \vdash\ (C)\, t : C} \quad \text{(T-WARN)}$$

$$\frac{\Gamma \vdash\ t : B, \quad B <: C}{\Gamma \vdash\ t : C} \quad \text{(T-SUB)}$$

Fig. 17. Type rules of gFJ (I)

Field and Method Lookup: $\Gamma \vdash \text{fields}(\mathsf{Object}) = \bullet$

$$\frac{\Gamma \vdash\ C \lessdot \langle\!\langle \mathsf{class}\ c\ \mathsf{extends}\ B\ \{\overline{FD}\ KD\ \overline{MD}\}\rangle\!\rangle}{\Gamma \vdash\ \text{fields}(C) = \text{fields}(B), \overline{FD}}$$

$$\Gamma \vdash\ \text{mtype}(\mathsf{Object}, m) = \bullet$$

$$\frac{\Gamma \vdash\ C \leq \langle\!\langle \mathsf{class}\ c\ \mathsf{extends}\ C'\ \{\overline{FD}\ KD\ \overline{MD}\}\rangle\!\rangle \quad B\ m(\overline{A}\ \overline{x})\ \{\mathsf{return}\ t;\} \in \overline{MD}}{\Gamma \vdash\ \text{mtype}(C, m) = \overline{A} \to B}$$

$$\frac{\Gamma \vdash\ C \leq \langle\!\langle \mathsf{class}\ c\ \mathsf{extends}\ B\ \{\overline{FD}\ KD\ \overline{MD}\}\rangle\!\rangle \quad m \notin \text{dom}(\overline{MD})}{\Gamma \vdash\ \text{mtype}(C, m) = \text{mtype}(B, m)}$$

***g*DEEP Subtyping:** $\boxed{\Gamma \vdash\ d \lhd d,\ \ \xi \lhd \xi}$

$$\frac{\Gamma \vdash\ \overline{B} \equiv \overline{C}}{\Gamma \vdash\ c(\overline{B}\ \overline{x})\ \{\mathsf{super}(\overline{t}); \overline{f} = \overline{u}\} \leq c(\overline{C}\ \overline{y})\ \{\mathsf{super}(\overline{t'}); \overline{f} = \overline{u'}\}}$$

$$\frac{\Gamma \vdash\ \overline{B} \equiv \overline{C}}{\Gamma \vdash\ \mathsf{refines}\ c(\overline{B}\ \overline{x})\ \{\overline{f} = \overline{u}\} \leq \mathsf{refines}\ c(\overline{C}\ \overline{y})\ \{\overline{f} = \overline{u'}\}}$$

$$\frac{\Gamma \vdash\ C \equiv C'}{\Gamma \vdash\ f\ C \equiv f\ C'}$$

$$\frac{\Gamma \vdash\ A \equiv A', \quad \overline{B} \equiv \overline{B'}}{\Gamma \vdash\ A\ m(\overline{B}\ \overline{x})\ \{\mathsf{return}\ t;\} \leq A'\ m(\overline{B'}\ \overline{y})\ \{\mathsf{return}\ u;\}}$$

Well-formedness: $\boxed{\Gamma \vdash\ \xi\ \text{wf},\ MD\ \text{wf}}$

$$\frac{\begin{array}{c}\Gamma \vdash\ \overline{A}\ \text{wf} \\ \Gamma \vdash\ C \lessdot \langle\!\langle \mathsf{class}\ c\ \mathsf{extends}\ B\{\overline{f}\ \overline{A'};\ KD\ \overline{MD}\}\rangle\!\rangle \\ \Gamma, \mathsf{this} : C, \overline{x} : \overline{A} \vdash\ \overline{u} : \overline{A'}, \quad \mathsf{new}\ B(\overline{t}) : B\end{array}}{\Gamma, X \leq C \vdash\ c(\overline{A}\ \overline{x})\ \{\mathsf{super}(\overline{t});\ \mathsf{this}.\overline{f} = \overline{u};\}\ \text{wf}}$$

$$\frac{\begin{array}{c}\Gamma \vdash\ A\ \text{wf}, \quad \overline{B}\ \text{wf} \\ \Gamma, \mathsf{this} : C, \overline{x} : \overline{B} \vdash\ t : A \\ \Gamma \vdash\ C \lessdot \langle\!\langle \mathsf{class}\ c\ \mathsf{extends}\ C'\{\overline{FD};\ KD\ \overline{MD}\}\rangle\!\rangle \\ \Gamma \vdash\ \text{mtype}(C', m)\ \text{is either}\ \overline{B} \to A\ \text{or}\ \bullet\end{array}}{\Gamma, X \leq C \vdash\ A\ m(\overline{B}\ \overline{x})\ \{\mathsf{return}\ t;\}\ \text{wf}}$$

Fig. 18. Type rules of gFJ (II)

premises of the form:

$$C \longrightarrow\!\!\!\!\!\rightarrow \langle\!\langle \mathsf{class}\ c\ \mathsf{extends}\ B\ \{\overline{FD}\ KD\ \overline{MD}\}\rangle\!\rangle$$

These premises use gDEEP to reduce the expression C to a value, which can be converted to a class declaration by using the inverse of the translation function $\langle\!\langle \_ \rangle\!\rangle$.

Typing is done similarly – the type rules in gFJ use the subtype rules provided by gDEEP in order to do modular type checking. The basic issue is the same: the type system needs to derive a class declaration CL from an expression C. gFJ also adds artifact- specific subtype and well-formedness rules that directly extend the corresponding judgements in gDEEP (see Fig. 18).

Typing differs from evaluation because during typing, a class expression C may contain free variables; e.g., it may be a local path of the form $X.c$. As a result, we cannot reduce C to a value. However, the subtype rules for gDEEP (i.e., the relations $\leq$, $\lessdot$ and $\equiv$) can be used to derive a minimal upper bound for C, using the judgement:

$$C \lessdot \langle\!\langle \mathsf{class}\ c\ \mathsf{extends}\ B\ \{\overline{FD}\ KD\ \overline{MD}\}\rangle\!\rangle$$

In some cases an upper bound is sufficient, so $\leq$ is used instead of $\lessdot$.

7 Integrating Further Artifact Languages

We illustrate how further artifact languages can be used with gDEEP, using Bali and XML as examples. Figure 19 shows the syntax of gBALI and gXML. In both cases, we have simplified the syntax for the purpose of clarity and presentation. The precise syntax of XML is both complex and irrelevant to the present discussion, and Bali similarly has more syntactic forms than those shown here.

Neither Bali nor Xak define their own compound modules. There are no Bali-specific or Xak-specific module annotations, and thus no need for a translation function. gDEEP records can be used as-is.

Bali grammars and XML documents are non-code artifacts. This means that there is no operational semantics, and no real type system. For both gBALI and gXML, we provide trivial rules for subtyping and well-formedness of artifact declarations.

There is really nothing more to define for gBALI and gXML other than the syntax. The only piece of syntax which needs to be interpreted and validated is delegation: terms of the form $M@(N).l$. The operational semantics and type system for gDEEP already provide reduction and well-formedness rules for delegation. Delegation is used similarly in both languages, as described below.

Bali. Rules in Bali can refer to other rules in the same grammar by name (i.e., l). They can also refer to the original definitions of rules in a base grammar (i.e., $\mathsf{original}.l$). Both of these terms are syntax sugar for standard gDEEP delegation – $M@(N).l$.

XML. A modular XML document assigns names to particular chunks of XML. It must be possible for one chunk of XML to refer to another chunk by name. It must also be possible for a named chunk of XML to refer to the original version of that chunk. Like gBALI, gXML uses standard gDEEP delegation to do both tasks.

***g*BALI Syntax:**		***g*XML Syntax:**	
a, b	literal characters	a, b	Literal characters
		g, h	Tag & attribute names
$d ::= t$	artifact declarations	$d ::= t$	Artifact declarations
$s, t, u ::=$	terms	$s, t, u ::=$	Terms (XML elements)
$M@(M).l$	rule path	$M@(M).l$	cross-reference
ϵ	empty string	$< g\ \overline{\alpha}\ />$	atomic XML node
'a'	character token	$< g\ \overline{\alpha} > \overline{\beta}\ </ c >$	compound XML node
"$\overline{a}$"	regular expression		
$t\ t$	concatenation	$\alpha ::=$	XML attribute
$t \mid t$	alternation	$g =$"$\overline{a}$"	
$t*$	zero or more		
$t+$	one or more	$\beta ::=$	XML node body
		$\overline{a}$	literal text
		t	XML element

Type rules for both systems:

$$\Gamma \vdash\ d \lhd d \quad \text{(S-ARTDECL)}$$

$$\Gamma \vdash\ d\ \text{wf} \quad \text{(W-ARTDECL)}$$

Fig. 19. Syntax and semantics of gBALI and gXML.

8 Implementation

In this work, we have been using *g*DEEP to explore the formal properties of features and feature composition. In a parallel line of research, we have been exploring how to take advantage of *g*DEEP in practical software development. We have developed a concrete representation of features, based on *g*DEEP, and a tool, called FSTCOMPOSE that composes features following the rules of *g*DEEP [38].

In a nutshell, we have a parser that generates a hierarchical language-independent representation of features (i.e., a concrete instance of *g*DEEP) that are written in Java. The representation is essentially an XML dialect that provides special tags for modeling compound and atomic modules. The tags contain proper type information (e.g., whether a module represents a class or a method) as well as the content of the modules and declarations of the artifact language.

Based on this representation, FSTCOMPOSE can compose a sequence of features, as defined by *g*DEEP's operator $\otimes$, and generate according Java code. The composition is performed entirely on the intermediate representation. Though tested with Java, FSTCOMPOSE is implemented generically. Further artifact languages can be integrated, presumed they obey a hierarchical module structure, as exposed by *g*DEEP. How to technically integrate further artifact languages, is explained elsewhere [38].

We have used our tool in three case studies to compose different Java programs (1–10 KLOC) of a set of features (8–88 features). The composition process has generated several tailored programs based on a user's feature selection. Our case studies have shown that our approach scales to medium-sized software projects: the average time of composition took less than two seconds for program sizes of up to 10 KLOC and 88 features. Further details about the case studies can be found elsewhere [38].

9 Related Work

*g*DEEP is inspired by DEEP [4], which is a formal object calculus that implements virtual classes [39] in a type-safe manner. The type system of DEEP is based on prototypes, which blur the distinction between objects and types. While DEEP works fine for Java-like software artifacts, it is overkill for artifacts like XML documents since there is no notion of computation in non-source code artifacts. We have stripped down DEEP to *g*DEEP. Several other calculi have been developed for virtual classes, e.g., *vc* [40], *Tribe* [41], *vObj* [42], *.FJ* [43]. All of them focus on the dynamic semantics and typing issues of virtual classes and thus are not appropriate for modeling multi-representation features.

It has been explored how features and feature composition can be expressed in terms of algebra [11,12,13]. The advantage of an algebra-based approach is that we can reason on an even more abstract level about features than *g*DEEP. The disadvantage is that the higher abstraction level of the algebra disallows formulating a precise definition of recursive hierarchical feature composition and of a general type system for FOP. We believe that both abstraction levels (calculus and algebra) are equally important exploring the principles of feature composition.

The notion of a feature is close to that of a component. The implementation a feature encapsulates several software artifacts, much like a component. However, multi-representation features consist of a diverse selection of artifacts of different types. Current component systems do not consider the composition of non-source code artifacts. It is not obvious how to extend or modify contemporary component and composition calculi (e.g., [44,45,46,47,48]) to incorporate artifacts of different types.

Several programming languages can be used to implement features in an FOP fashion, e.g., *CaesarJ* [20], *Hyper/J* [22], *Classbox/J* [21], *Jx* [23], *Scala* [25], *Jiazzi* [24], *FeatureC++* [10], *Jak* [2]. However, only FeatureC++, Jak, and Jiazzi separate the notion of a feature (collaboration) and the underlying artifact-specific host language (C++ and Java). Thus, they can be used for the implementation of multi-representation features. *g*DEEP is a calculus to treat them uniformly and is a basis for a tool suite that composes artifacts written in these languages.

Several tools support the subsequent refinement and the composition of code and non-code artifacts, e.g., feature equations [2], feature diagrams [29], syntax specifications [30], UML models [31,32], and aspects [6]. All of them could be part of a multi-representation feature and formalized in *g*DEEP.

Several case studies on AHEAD demonstrated the necessity of managing multi-representation features [2,49,28,9] and work on SPLs and component models (e.g., CORBA, DCOM) in general suggests that a software product usually consists of many different artifacts (a.k.a. assets) [50]. Another branch of research explores the possibilities of interaction between artifacts written in different languages [51], which would be the next logical step in implementing multi-representation features and their formalization in *g*DEEP.

10 Conclusion

The principle of uniformity states that features are implemented typically by multiple types of software artifacts, and these artifacts are subject of subsequent refinement. We have developed *g*DEEP as a formal core calculus that encapsulates the essence of feature composition and refinement. It abstracts from artifact-specific details and treats any kind of software artifact in the same way. We have presented a formal syntax, operational semantics, and type system of *g*DEEP and illustrated what a language needs to provide when it is plugged into *g*DEEP and how languages are prepared in case they are not "feature-ready". We have adapted and developed formalizations of Java, Bali, and XML and plugged them into the core calculus. This demonstrates the generality of expressivness of *g*DEEP.

Our calculus serves also as an intermediate language for feature representation and manipulation, which is a foundation for large-scale feature-oriented program synthesis [3]. We have a tool that implements feature composition following the principles of *g*DEEP. Three case studies demonstrate its practicality and scalability. Further algorithms and tools can be developed on top of the calculus to provide a seamless infrastructure for developing, managing, composing, and validating features in different representations.

Acknowledgments

We thank Don Batory, Christian Kästner, Christian Lengauer, and Marko Rosenmüller for their helpful comments on earlier drafts of this paper. Furthermore, we thank Don Batory for sharing his idea about hierarchical type systems for FOP.

References

1. Prehofer, C.: Feature-Oriented Programming: A Fresh Look at Objects. In: Proceedings of European Conference on Object-Oriented Programming. Volume 1241 of LNCS., Springer (1997) 419–443
2. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. on Software Engineering **30** (2004) 355–371
3. Batory, D.: From Implementation to Theory in Product Synthesis. In: Proceedings of International Symposium. on Principles of Programming Languages, ACM Press (2007) 135–136
4. Hutchins, D.: Eliminating Distinctions of Class: Using Prototypes to Model Virtual Classes. In: Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (2006) 1–19
5. Apel, S., Leich, T., Saake, G.: Aspectual Mixin Layers: Aspects and Features in Concert. In: Proceedings of International Conference on Software Engineering, ACM Press (2006) 122–131
6. Apel, S., Kästner, C., Leich, T., Saake, G.: Aspect Refinement - Unifying AOP and Stepwise Refinement. Journal of Object Technology – Special Issue: TOOLS EUROPE 2007 **6** (2007) 13–33
7. Apel, S., Kästner, C., Kuhlemann, M., Leich, T.: Pointcuts, Advice, Refinements, and Collaborations: Similarities, Differences, and Synergies. Innovations in Systems and Software Engineering – A NASA Journal **3** (2007)
8. Apel, S.: The Role of Features and Aspects in Software Development. PhD thesis, School of Computer Science, University of Magdeburg (2007)
9. Apel, S., Batory, D.: When to Use Features and Aspects? A Case Study. In: Proceedings of International Conference on Generative Programming and Component Engineering, ACM Press (2006) 59–68
10. Apel, S., Leich, T., Rosenmüller, M., Saake, G.: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In: Proceedings of International Conference on Generative Programming and Component Engineering. Volume 3676 of LNCS., Springer (2005) 125–140
11. Lopez-Herrejon, R., Batory, D., Cook, W.: Evaluating Support for Features in Advanced Modularization Technologies. In: Proceedings of European Conference on Object-Oriented Programming. Volume 3586 of LNCS., Springer (2005) 169–194
12. Lopez-Herrejon, R., Batory, D., Lengauer, C.: A Disciplined Approach to Aspect Composition. In: Proceedings of International Symposium. on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press (2006) 68–77
13. Apel, S., Lengauer, C., Batory, D., Möller, B., Kästner, C.: An Algebra for Feature-Oriented Software Development. Technical Report MIP-0706, Department of Informatics and Mathematics, University of Passau (2007)
14. Ossher, H., Harrison, W.: Combination of Inheritance Hierarchies. In: Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (1992) 25–40

15. Findler, R., Flatt, M.: Modular Object-Oriented Programming with Units and Mixins. In: Proceedings of International Conference on Functional Programming, ACM Press (1998) 94–104
16. Smaragdakis, Y., Batory, D.: Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM Trans. on Software Engineering and Methodology **11** (2002) 215–255
17. Bouge;, L., Francez, N.: A Compositional Approach to Superimposition. In: Proceedings of International Symposium. on Principles of Programming Languages, ACM Press (1988) 240–249
18. Bosch, J.: Super-Imposition: A Component Adaptation Technique. Information and Software Technology **41** (1999) 257–273
19. Ernst, E.: Higher-Order Hierarchies. In: Proceedings of European Conference on Object-Oriented Programming. Volume 2743 of LNCS., Springer (2003) 303–329
20. Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: An Overview of CaesarJ. Trans. on Aspect-Oriented Software Development **1** (2006) 135–173
21. Bergel, A., Ducasse, S., Nierstrasz, O.: Classbox/J: Controlling the Scope of Change in Java. In: Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (2005) 177–189
22. Ossher, H., Tarr, P.: Hyper/J: Multi-Dimensional Separation of Concerns for Java. In: Proceedings of International Conference on Software Engineering, IEEE CS Press (2000) 734–737
23. Nystrom, N., Chong, S., Myers, A.: Scalable Extensibility via Nested Inheritance. In: Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (2004) 99–115
24. McDirmid, S., Flatt, M., Hsieh, W.: Jiazzi: New-Age Components for Old-Fashioned Java. In: Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (2001) 211–222
25. Odersky, M., Zenger, M.: Scalable Component Abstractions. In: Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (2005) 41–57
26. Costanza, P., Hirschfeld, R., de Meuter, W.: Efficient Layer Activation for Switching Context-Dependent Behavior. In: Proceedings of Joint Modular Languages Conference. Volume 4228 of LNCS., Springer (2006) 84–103
27. Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: Tools for Implementing Domain-Specific Languages. In: Proceedings of International Conference on Software Reuse, IEEE CS Press (1998) 143–153
28. Anfurrutia, F., Díaz, O., Trujillo, S.: On Refining XML Artifacts. In: Proceedings of International Conference on Web Engineering. Volume 4607 of LNCS., Springer (2007) 473–478
29. Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C.: Refactoring Product Lines. In: Proceedings of International Conference on Generative Programming and Component Engineering, ACM Press (2006) 201–210
30. Bravenboer, M., Visser, E.: Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation Without Restrictions. In: Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (2004) 365–383
31. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Proceedings of International Conference on Generative Programming and Component Engineering. Volume 3676 of LNCS., Springer (2005) 422–437
32. Laguna, M., González-Baixauli, B., Marqués, J.: Seamless Development of Software Product Lines. In: Proceedings of International Conference on Generative Programming and Component Engineering, ACM Press (2007) 85–94

33. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
34. Thaker, S., Batory, D., Kitchin, D., Cook, W.: Safe Composition of Product Lines. In: Proceedings of International Conference on Generative Programming and Component Engineering, ACM Press (2007) 95–104
35. Liu, J., Batory, D., Lengauer, C.: Feature-Oriented Refactoring of Legacy Applications. In: Proceedings of International Conference on Software Engineering, ACM Press (2006) 112–121
36. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ. In: Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (1999) 132–146
37. Cardelli, L., Wegner, P.: On Understanding Types, Data Abstraction, and Polymorphism. ACM Comp. Surv. **17** (1985) 471–522
38. Apel, S., Lengauer, C.: Superimposition: A Language-Independent Approach to Software Composition. Technical Report MIP-0711, Department of Informatics and Mathematics, University of Passau (2007)
39. Madsen, O., Moller-Pedersen, B.: Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In: Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (1989) 397–406
40. Ernst, E., Ostermann, K., Cook, W.: A Virtual Class Calculus. In: Proceedings of International Symposium. on Principles of Programming Languages, ACM Press (2006) 270–282
41. Clarke, D., Drossopoulou, S., Noble, J., Wrigstad, T.: Tribe: A Simple Virtual Class Calculus. In: Proceedings of International Conference on Aspect-Oriented Software Development, ACM Press (2007) 121–134
42. Odersky, M., Cremet, V., Röckl, C., Zenger, M.: A Nominal Theory of Objects with Dependent Types. In: Proceedings of European Conference on Object-Oriented Programming. Volume 2743 of LNCS., Springer (2003) 201–224
43. Igarashi, A., Saito, C., Viroli, M.: Lightweight Family Polymorphism. In: Proceedings of Asian Symposium. on Programming Languages and Systems. Volume 3780 of LNCS., Springer (2005) 161–177
44. Achermann, F., Nierstrasz, O.: A Calculus for Reasoning About Software Composition. Theoretical Computer Science **331** (2005) 367–396
45. Pucella, R.: Towards a Formalization for COM Part I: The Primitive Calculus. In: Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (2002) 331–342
46. Zenger, M.: Type-Safe Prototype-Based Component Evolution. In: Proceedings of European Conference on Object-Oriented Programming. Volume 2374 of LNCS., Springer (2002) 470–497
47. Seco, J., Caires, L.: A Basic Model of Typed Components. In: Proceedings of European Conference on Object-Oriented Programming. Volume 1850 of LNCS., Springer (2000) 108–128
48. Vion-Dury, J.Y., Bellissard, L., Marangozov, V.: A Component Calculus for Modeling the Olan Configuration Language. In: Proceedings of International Conference on Coordination Languages and Models. Volume 1282 of LNCS., Springer (1997) 392–409
49. Trujillo, S., Batory, D., Díaz, O.: Feature Refactoring a Multi-Representation Program into a Product Line. In: Proceedings of International Conference on Generative Programming and Component Engineering, ACM Press (2006) 191–200
50. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2002)

51. Grechanik, M., Batory, D., Perry, D.: Design of Large-Scale Polylingual Systems. In: Proceedings of International Conference on Software Engineering, IEEE CS Press (2004) 357–366