A Calculus for Uniform Feature Composition

SVEN APEL
University of Passau
and
DELESLEY HUTCHINS
MZA Associates Corporation

The goal of *feature-oriented programming* (FOP) is to modularize software systems in terms of features. A *feature* refines the content of a base program. Both base programs and features may contain various kinds of software artifacts, for example, source code in different languages, models, build scripts, and documentation. We and others have noticed that when composing features, different kinds of software artifacts can be refined in a uniform way, regardless of what they represent. We present *g*DEEP, a core calculus for feature composition, which captures the language independence of FOP; it can be used to compose features containing many different kinds of artifact in a type-safe way. The calculus allows us to gain insight into the principles of FOP and to define general algorithms for feature composition and validation. We provide the formal syntax, operational semantics, and type system of *g*DEEP and outline how languages like Java, Haskell, Bali, and XML can be plugged in.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: Feature-oriented programming, feature composition, type systems, principle of uniformity

ACM Reference Format:
Apel, S. and Hutchins, D. 2010. A calculus for uniform feature composition. ACM Trans. Program. Lang. Syst. 32, 5, Article 19 (May 2010), 33 pages.
DOI = 10.1145/1745312.1745316 http://doi.acm.org/10.1145/1745312.1745316

This work was funded in part by the German Research Foundation (DFG), project number AP 206/2-1.
Authors' addresses: S. Apel, Department of Informatics and Mathematics, University of Passau, Innstr. 33, 94032 Passau, Germany; email: apel@uni-passau.de; D. Hutchins, MZA Associates Corporation, 2021 Girard Blvd. SE, Suite 150, Albuquerque, NM 87106-3140; email: delesley@gmail.com.

DOI 10.1145/1745312.1745316 http://doi.acm.org/10.1145/1745312.1745316

1. INTRODUCTION

The goal of *feature-oriented programming* (FOP) is to modularize software systems in terms of features [Prehofer 1997]. A *feature* is a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option [Apel and Kästner 2009]. *AHEAD* (*Algebraic Hierarchical Equations for Application Design*) is an architectural model for large-scale FOP [Batory et al. 2004].

The idea behind AHEAD is to unify 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 base program is modeled as a collection of named program elements, which are organized in a hierarchical namespace. 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. The hierarchical namespace allows a refinement to target any element at any depth of the hierarchy.

Second, AHEAD scales program refinement to arbitrary kinds of software artifacts. A feature typically includes changes not only to the source code, but also to other supporting documents, for example, HTML documentation, ANT scripts, and UML diagrams. The principle of uniformity that underlies AHEAD can be stated as follows: features are implemented by a diverse selection of software artifacts, and any kind of software artifact can be the subject of subsequent refinement [Batory et al. 2004].

Whereas the idea of uniform feature composition captures the philosophy behind the AHEAD model and guides us when reasoning about feature-based program synthesis [Batory 2007], it is rather abstract. Although our previous work on FOP [Hutchins 2006, 2009; Apel et al. 2005, 2008b, 2009c, 2009a, 2009b; Apel and Lengaver 2008] had established that program refinement is similar for all artifact types, we had no way to express and reason about this similarity. We wanted a formal model of feature composition that was independent of the particular kind of software artifact that was being refined. Such a model answers several important questions: What is the essence of feature-based 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 will help us answer these questions.

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

(1) It enables us to reason about the properties and procedures of program refinement and feature composition in a formal way.
(2) Software artifacts of different types can be plugged into the calculus and treated equally by the algorithms for feature composition and validation. The algorithms can be expressed in a uniform and language-independent way.

(3) *g*DEEP's type system factors out the portion of typing that is concerned with feature composition. Rather than designing separate and incompatible type systems to handle features in each artifact language, we can use a single type system. The type system for features integrates with the type system of each artifact language in a uniform way.
(4) Tools can operate directly on a concrete representation of *g*DEEP. This way, tools for composition, validation, and analysis of features can be reused for various types of artifacts.

*g*DEEP builds on the seminal work on mixin composition [Bracha and Cook 1990; Findler and Flatt 1998; Flatt et al. 1998; Bono et al. 1999; Smaragdakis and Batory 2002; Ancona et al. 2003] and generalizes and scales previous work on a formal foundation for FOP [Hutchins 2006]. It is an alternative to approaches that rely on algebra [Lopez-Herrejon et al. 2006; Apel et al. 2008c, 2010b] and complements work on tools and case studies [Apel and Lengauer 2008; Apel et al. 2009c], which is discussed in Section 8. We present the syntax, operational semantics, and type system of *g*DEEP and explain how different artifact languages such as Java, Haskell, Bali, and XML can be plugged into it. We focus here on the key design decisions and the generality of *g*DEEP and not on the underlying type theory. Although this article is self-contained, in some paragraphs, we refer to an accompanying technical report that provides more details [Apel and Hutchins 2007].

2. FEATURE-ORIENTED PROGRAMMING

A feature refines the content of a base program either by 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 can be composed with other features by function composition, or composed with a base program by applying the function to yield another program. The order in which features are applied is important; earlier features in the sequence may add elements that are refined by later features. Existing FOP tools perform feature composition at compile-time, although this is not a requirement.

AHEAD is an architectural model of FOP [Batory et al. 2004]. With AHEAD, each feature is implemented by a *containment hierarchy*, which is a directory that maintains a substructure 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 [Ossher and Harrison 1992; Tarr et al. 1999], mixin composition [Bracha and Cook 1990; Findler and Flatt 1998; Flatt et al. 1998; Smaragdakis and Batory 2002; Bono et al. 1999; Ancona et al. 2003], and superimposition [Bouge and Francez 1988; Bosch 1999]. In contrast to these earlier approaches, for each artifact type, a different implementation of the *composition operator* '•', that is, a tool that performs the composition, has to be provided in AHEAD.

Feature *Expr*

```
1  abstract class Expr {
2    abstract String toString();
3  }
4
5  class Val extends Expr {
6    int val;
7    Val(int n) { val = n; }
8    String toString() {
9      return String.valueOf(val);
10   }
11 }
12
13 class Add extends Expr {
14   Expr a; Expr b;
15   Add(Expr e1, Expr e2) { a = e1; b = e2; }
16   String toString() {
17     return a.toString() + "+" + b.toString();
18   }
19 }
```

Feature *Eval* refines *Expr*

```
20 refines class Expr {
21   abstract int eval();
22 }
23
24 refines class Val {
25   int eval() { return val; }
26 }
27
28 refines class Add {
29   int eval() {
30     return a.eval() + b.eval();
31   }
32 }
```

Feature *Mult* refines *Expr*

```
33 class Mult extends Expr {
34   Expr a; Expr b;
35   Mult(Expr e1, Expr e2) { a = e1; b = e2; }
36   String toString() {
37     return "(" + a.toString() +
38            "*" + b.toString() + ")";
39   }
40 }
41
42 refines class Add {
43   String toString() {
44     return "(" + Super.toString() + ")";
45   }
46 }
```

Derivative *Mult* + *Eval*

```
47 refines class Mult {
48   int eval() {
49     return a.eval() * b.eval();
50   }
51 }
```

Feature *Var* refines *Expr*

```
52 class Env implements Map<String, Expr> {...};
53
54 refines class Expr {
55   abstract Expr replaceVars(Env e);
56 }
57
58 class Var extends Expr {
59   String name;
60   Var(String n) { name = n; }
61   Expr replaceVars(Env e) {
62     if (e.containsKey(name))
63       return e.get(name);
64     else return new Var(name);
65   }
66 }
67
68 refines class Val {
69   Expr replaceVars(Env e) {
70     return new Val(val);
71   }
72 }
73
74 refines class Add {
75   Expr replaceVars(Env e) {
76     return new Add(a.replaceVars(e),
77                    b.replaceVars(e));
78   }
79 }
```

Derivative *Var* + *Eval*

```
80 refines class Var {
81   int eval() {
82     throw new Exception("Unknown variable");
83   }
84 }
```

Derivative *Var* + *Mult*

```
85 refines class Mult {
86   Expr replaceVars(Env e) {
87     return new Mult(a.replaceVars(e),
88                     b.replaceVars(e));
89   }
90 }
```

main function, using all features

```
91 class Main {
92   static void main(String[] args) {
93     Expr e1 = new Add(new Var("x"),
94                       new Val(1));
95     Env env = new Env();
96     env.put("x", 1);
97     Expr e2 = e1.replaceVars(env);
98     System.out.println(
99       e1.toString() + " = " + e2.eval()
100    );
101  }
102 }
```

Fig. 1. A solution to the "expression problem" in Jak.

2.1 Jak

Jak is an implementation of a composition operator for Java artifacts [Batory et al. 2004]. Figure 1 depicts the Jak code of an expression evaluator, which is a

feature-oriented solution to the well-known "expression problem".[1] The expression problem has two elements that are commonly found in most programs:

(1) A recursively-defined variant data type (e.g., an abstract syntax tree).
(2) A set of recursive operations over that data type (e.g., evaluation and pretty printing).

A solution to the expression problem is a programming language or technique that satisfies the following three requirements:

(1) It is possible to extend the data type with new variants.
(1) It is possible to add new operations.
(2) Different extensions can be defined separately, and then later combined.

The expression problem illustrates a broader issue called the "tyranny of the dominant decomposition" [Tarr et al. 1999]. In most programming languages, it is possible to extend either the data type or the set of operations, but not both at the same time. The data type and the operations are two different *concern dimensions*. Although both dimensions are conceptually of equal importance, existing languages require code to be factored in such a way that one is prioritized over the other. In an object-oriented language, the interpreter design pattern prioritizes the data type, and thus allows new variants to be easily added, whereas the visitor design pattern prioritizes the operations. No matter how the code is factored, concerns belonging not to the "dominant" dimension cut across the implementations of concerns belonging to other dimensions [Tarr et al. 1999; Kiczales et al. 1997]. The implementation of such a "crosscutting concern" is scattered throughout the code, where it is difficult trace for the programmer and difficult to extend with standard programming language constructs.

Feature-oriented programming tools, such as Jak, provide a mechanism that allows crosscutting concerns to be defined separately and then mixed together. Figure 1 follows the interpreter design pattern, in which variants are classes, with a method for each operation. It splits the functionality into four separate features.

Feature *Expr* represents the base program. It defines class Expr, along with two variants: Val for integer literals and Add for addition. It also defines a single operation toString for pretty printing. Feature *Eval* adds the new operation eval, which evaluates an expression. Evaluation is a crosscutting concern because eval must be defined by adding a method to each of the three classes. Feature *Mult* adds the new variant Mult. Adding a new variant is not ordinarily a crosscutting concern. However, once the language supports both multiplication and addition, the precedence of operations becomes important. Thus, feature *Mult* also extends pretty printing in order to add parentheses. The new version of Add.toString calls the old version using Jak's keyword Super. Feature *Var*

[1]The expression problem was named by Phil Wadler in 1998 but has been known for many years [Reynolds 1994; Cook 1991; Krishnamurthi et al. 1998]; see Torgersen [2004] for a retrospective overview.

adds both a new variant Var, which implements named variables, and a new operation replaceVars, which replaces all the variables in an expression with their definitions.

The features *Eval*, *Mult*, and *Var* are each designed to extend *Expr*. However, they are not completely orthogonal. The combination of a new variant and a new operation, creates a "missing piece" that must be filled in to create a complete program. We thus define three additional features, called *lifters* [Prehofer 1997] and *derivatives* [Liu et al. 2006; Kästner et al. 2009], that define how each feature should be extended in the presence of the others. For example, the derivative '*Mult* + *Eval* is present when both features *Mult* and *Eval* are present.

2.2 Jak & Virtual Classes

It is worth examining briefly why the features in Figure 1 cannot simply be implemented with standard object-oriented inheritance. The most obvious problem is that feature composition would require multiple inheritance, which Java does not have. However, there is a more important and more subtle issue involved.

Object-oriented inheritance creates a new class with a different name. Thus, in order to implement feature *Eval* using standard object-oriented inheritance, we would have to create three new classes: ExprEval, ValEval, and AddEval. Creating new classes with different names would break the implementation of other features, because all references to the original class names (e.g., Expr and Add) would refer to the old definitions instead of the new ones. Classes are referred to by name in two places: (1) when constructing instances of the class and (2) in static type information.

Constructors. Feature *Var* copies an expression by calling the constructors of Val and Add. If these constructors referred to the original class definitions, then copying would produce an expression that did not support any of the operations defined in other features, such as eval. When composing features with Jak, this problem does not occur; notice that the main routine calls replaceVars followed by eval.

Typing. The classes Add and Mult have data members of type Expr. Operations such as eval and replaceVars are defined recursively on these members. Thus, these additional operations must be defined within class Expr itself in order for the implementation to be well-typed; they cannot be defined within derived classes.

Unlike object-oriented inheritance, the Jak tool does not create new classes with different names. Instead, each feature defines a different slice (or role) of class Expr. The Jak tool collects the different slices and merges them together into a single class definition. In essence, Jak implements *virtual classes* [Madsen and Moller-Pedersen 1989] and *family polymorphism* [Ernst 2001]. With FOP, both methods and classes use *late binding*. Classes can be refined by subsequent features in the same way that methods are overridden during standard object-oriented inheritance. Class names like Expr and Add are not resolved until after feature composition has taken place.

Feature *Expr*

```
1 Expr: Val | Expr Oper Expr;
2 Oper: '+';
3 Val: INTEGER;
```

Feature *Mult* refines *Expr*

```
4 Oper: Super.Oper | '*';
```

Feature *Var* refines *Expr*

```
5 Expr: Super.Expr | Var | 'let' Var '=' Expr 'in' Expr;
6 Var: IDENTIFIER;
```

Fig. 2. A Bali grammar with separate features for addition, multiplication, variables, and evaluation.

As Krishnamurthi et al. point out, the problem of class construction can be resolved by using virtual factory methods, at the cost of some additional coding effort [Krishnamurthi et al. 1998]. However, the typing issue is not so easily resolved and requires a type system that can handle virtual types [Ernst et al. 2006].

As a side note, we wish to point out that FOP is slightly less flexible than the family polymorphism in gbeta [Ernst 2001]. In gbeta, a family of classes is encapsulated within an object, which exists at run-time. In contrast, Jak performs feature composition at compile-time. Features are "erased" from the final compiled code, and are not reified as run-time objects. The advantage of the erasure semantics is that it allows FOP to be used with languages, like Java, that do not have built-in support for features. The disadvantage is that, unlike gbeta, it is not possible to instantiate new families at run-time.

2.3 Bali

A complete software system does not just involve Java code. It also involves many non-code artifacts. For example, the simple expression evaluator in Figure 1 may be paired with a grammar specification that provides a concrete syntax for expressions.

Bali is a tool for synthesizing program manipulation tools on the basis of extensible grammar specifications [Batory et al. 2004]. It allows a programmer to define a grammar and to refine it subsequently, in a similar fashion to class refinements in Jak. Figure 2 shows a grammar and two grammar refinements that correspond to the Jak program above. The base program defines the syntax of arithmetic expressions that involve addition only. We then refine the grammar by adding support for multiplication and variables.

Unlike other tools for grammar specification, the grammar rules in Bali use late binding: the name Expr is not resolved until after feature composition. Bali is also similar to Jak in its use of keyword Super: Expression Super.Oper refers to the original definition of Oper.

2.4 Xak

Semistructured program documentation is another example of a non-code artifact. *Xak* is a language and tool for composing various kinds of XML

Feature *Expr*

```
<html xmlns:xak="http://www.onekin.org/xak" xak:artifact="Expr" xak:type="xhtml">
  <head><title>A Simple Expression Evaluator</title></head>
  <body bgcolor="white">
    <h1 xak:module="Contents">A Simple Expression Evaluator</h1>
    <h2>Supported Operations</h2>
    <ul xak:module="Operations">
      <li>Addition of integers</li>
      <!-- a description of how integers are added -->
    </ul>
  </body>
</html>
```

Feature *Eval* refines *Expr*

```
<xak:refines xmlns:xak="http://www.onekin.org/xak" xak:artifact="Eval" xak:type="xhtml">
  <xak:extends xak:module="Contents">
    <xak:super xak:module="Contents"/>
    <h2>Evaluation of Arithmetic Expressions</h2>
    <!-- a description of how expressions are evaluated -->
  </xak:extends>
</xak:refines>
```

Feature *Mult* refines *Expr*

```
<xak:refines xmlns:xak="http://www.onekin.org/xak" xak:artifact="Mult" xak:type="xhtml">
  <xak:extends xak:module="Operations">
    <xak:super xak:module="Operations"/>
    <li>Multiplication of integers</li>
    <!-- a description of how integers are multiplied -->
  </xak:extends>
</xak:refines>
```

Feature *Var* refines *Expr*

```
<xak:refines xmlns:xak="http://www.onekin.org/xak" xak:artifact="Var" xak:type="xhtml">
  <xak:extends xak:module="Contents">
    <xak:super xak:module="Contents"/>
    <h2>Support for Variables</h2>
    <!-- a description of how variables are handled -->
  </xak:extends>
</xak:refines>
```

Fig. 3. A Xak/XHTML document with separate features for addition, evaluation, multiplication, and variables.

documents [Anfurrutia et al. 2007]. It enhances XML by a module structure useful for refinement. This way, a broad spectrum of software artifacts can be refined à la FOP, e.g., UML diagrams (XMI), build scripts (ANT), service interfaces (WSDL), server pages (JSP), or XHTML.

Figure 3 depicts an XHTML document that contains documentation for our expression evaluator. The base documentation file describes addition only, but we refine it in a mixin style to add a description of evaluation, multiplication, and variables as well. The tag xak:module labels a particular XML element with a name that allows the element to be refined by subsequent features. The tag xak:extends overrides an element that has been named previously, and the tag xak:super refers to the original definition of the named element, just like the keyword Super in Jak and Bali.

Feature composition tools like Xak require the software artifact to have a named hierarchical structure. As far as documentation is concerned, FOP works well for semistructured text such as websites, reference manuals, and API documentation [Trujillo et al. 2006; Apel et al. 2009c]. FOP is less useful for

narrative documents like novels, because features may not have proper points to "hook in" and feature composition may disrupt the flow of text by inserting sections.

2.5 AHEAD

Jak, Xak, and Bali are each designed to work with a particular kind of software artifact. AHEAD brings these separate tools together into a system that can handle many different kinds of software artifacts.

In AHEAD, a piece of software is represented as a directory of files. Composing two directories together will merge subdirectories and files with the same name. AHEAD will select different composition tools for different kinds of files. Merging Java files will invoke Jak to refine the classes, whereas merging XML files will invoke Xak to combine the XML documents, and so on.

Recently, following the philosophy of AHEAD, the FeatureHouse tool suite has been developed that allows programmers to enhance given languages rapidly with support for FOP, for example, C#, C, JavaCC, Haskell, Alloy, and UML [Apel et al. 2009c].

3. AN INFORMAL OVERVIEW OF *g*DEEP

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

We call our calculus *g*DEEP, which is short for *generalized* DEEP. As its name might suggest, *g*DEEP is based on the DEEP calculus. The main concepts of DEEP were initially presented in an earlier paper [Hutchins 2006], and are explored in detail in the Ph.D. dissertation of Hutchins [2009]. The DEEP calculus provides two capabilities that are relevant to FOP.

First, DEEP provides a formal model of deep mixin composition [Zenger and Odersky 2005; Hutchins 2009], which is related to a number of techniques in the literature, including higher-order hierarchies [Ernst 2003], virtual classes [Madsen and Moller-Pedersen 1989], nested inheritance [Nystrom et al. 2004], and superimposition [Apel and Lengauer 2008]. The "deep" part of composition scales object-oriented inheritance to programming in the large. Unlike inheritance in Java or C#, which support only the overriding of virtual methods, the DEEP calculus supports the refinement (i.e., overriding) of nested classes and submodules within a module hierarchy. The "mixin" part of composition allows separate refinements to be combined together, in a fashion that is somewhat analogous to multiple inheritance [Bracha and Cook 1990].

Second, DEEP provides a static type system that can handle deep mixin composition. This is not so important for untyped artifacts such as XML, but it is important for typed artifacts such as Bali and Java, since it addresses a crucial weakness of existing feature-oriented tools. In existing tools, all type checking must be done after composition. Using the DEEP type system, we can type check features before composition. Note that we do not provide proofs or

cover the metatheory of DEEP in this paper; readers interested in technical details should refer to Hutchins' Ph.D. dissertation [Hutchins 2009].

The original DEEP calculus was conceived as a formal model of a stand-alone programming language. The main contribution of this article is to demonstrate that the module system of DEEP can be used for a wide variety of different artifact languages. The *g*DEEP calculus extends DEEP with hooks that allow us to embed Java programs, XML documents, and so on within DEEP modules.

3.1 Deep Mixin Composition

Features in *g*DEEP are represented as either modules or functions over modules. A base feature is an ordinary module, whereas a feature that performs a refinement is a function that transforms a module. Modules have a named hierarchical structure; a module may contain submodules, subsubmodules, and so on.

Code in the artifact language can be represented in one of two ways: *Compound declarations* are modules, which means they have a named substructure. For example, a Java class is compound because it contains named methods and nested classes. *Atomic declarations* do not have a particular substructure which *g*DEEP can interpret; an atomic declaration can be any arbitrary term in the artifact language. Examples of atomic declarations are Java methods and fields, ordinary XML declarations, and Bali grammar rules.

Modules are composed by recursively composing their constituent declarations. The composition of two modules A and B will have all the declarations of A and all the declarations of B. If A and B both have a declaration with the same name, then those two declarations will be composed recursively. Composition thus descends into the module hierarchy and recursively merges submodules together until it reaches the atomic declarations, which form the leaves of the hierarchy.

The composition of two atomic declarations works much like standard object-oriented method overriding. One declaration overrides the content of the other. However, the overriding declaration may use the metavariable original (much like Java's super or Jak's Super) to refer to the overridden content. By using keyword original it is possible to combine the content of two atomic declarations together without needing to know details about them. The calculus assumes only that the artifact language is defined using terms and that a substitution operator is available over terms.

3.2 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 declaration in *g*DEEP.

Bali is a simple artifact language, because it does not have any compound elements, that is, there are no nested grammar production rules [Batory et al. 2004]. As a result, it is very easy to plug it into *g*DEEP. The *g*DEEP calculus provides a hierarchical module system, and all expressions in the artifact language become atomic declarations. The module calculus and the artifact language are

almost completely orthogonal.

XML elements may have subelements that are subject to refinement. Xak explicitly establishes a module structure for XML by associating a module name with a particular XML element using the attribute xak:module [Anfurrutia et al. 2007]. Thus, the module structure defined by Xak is largely independent of the structure of the XML document itself. XML elements that are tagged with xak:module become modules, whereas ordinary elements become atomic declarations. Once again, the module language and artifact language are largely orthogonal.

Java is the most difficult to handle, because Java and *g*DEEP are not orthogonal. Java defines different kinds of compound elements: packages contain named subpackages, classes, and interfaces, whereas classes contain named inner classes, methods, and fields, and so on. Hence, we map packages, classes, and interfaces onto modules in *g*DEEP. Methods and fields have no named substructure, so we represent them as atomic declarations.

We handle compound elements of the artifact language by defining a translation function. In the case of Java, the translation function provides a bijective mapping between Java classes and *g*DEEP modules. In the case of XML, the translation function provides a bijective mapping between Xak modules and *g*DEEP modules. Since the translation function is bijective, any manipulations performed within *g*DEEP can be translated back to the artifact language.

Once a software artifact has been translated to *g*DEEP, we can use the calculus to compose features in a type-safe and language-independent way. Other languages such as JavaCC, C++, C#, or C are plugged into *g*DEEP in similar ways.

3.3 Constraints on the Artifact Language

Feature composition with *g*DEEP imposes three constraints on the artifact language:

(1) The substructure of a feature must be a hierarchy of modules.
(2) Every module (submodule, ...) of a feature must have a name.
(3) The name of a module must be unique in the scope of its enclosing module. That is, a module must not have two submodules with identical names.

These constraints are satisfied by most programming languages. In addition, many other (noncode) languages align well with them [Batory et al. 2004; Anfurrutia et al. 2007; Apel and Lengauer 2008; Apel et al. 2005, 2009c], as we will illustrate in Section 7.

Languages that do not satisfy these constraints do not provide sufficient structural information for a feature composition with *g*DEEP. For example, plain XHTML allows children of an XML element to be identical, for example, the elements of a list. However, these languages may be enriched by assigning unique names or by providing an overlaying module structure, for example, as Xak does for XML [Anfurrutia et al. 2007].

3.4 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 based on the type system of DEEP. *g*DEEP's type system provides type judgments at the module/feature level that are universal over all supported artifact types. The *g*DEEP type system has two main capabilities:

First, there is a subtype relation defined over features and feature compositions. Features which require other features can express that dependency by means of subtyping (see Section 4.6). The type checker will ensure that all dependencies are properly declared as requirements and that all requirements of a composition are satisfied.

Second, it is possible to use artifact-specific type systems in concert with the *g*DEEP type system. For example, the standard Java type system is responsible for assigning types to Java expressions, which always occur in atomic modules. The type of an expression in Java is the name of a class (or a basic type). We integrate the Java type system into *g*DEEP by replacing the default Java class lookup mechanism. The *g*DEEP type system becomes responsible for looking up class and method names within features, since such names cross module boundaries (see Section 6 for more details). The core theorems of DEEP ensure that this combination is sound. However, a discussion of DEEP's core theorems is out of scope, and we refer the reader to Hutchins [2006, 2009].

Current feature composition tools, such as Jak, do not support modular type checking; 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.

4. OVERVIEW OF THE *g*DEEP CORE CALCULUS

The module system of *g*DEEP is a generalization of DEEP. The DEEP calculus is described elsewhere in greater detail [Hutchins 2009]; we begin with a brief overview of DEEP and proceed with *g*DEEP's syntax and semantics.

Modules in DEEP were designed with the following goals in mind.

—Modules may contain both type-level definitions (e.g., classes) and object-level definitions (e.g., methods and data members).

—Modules are recursive. One definition within a module can refer to other definitions by name, in mutually recursive ways. Module-level recursion can be used to define both recursive types (e.g., classes like List) and recursive objects (e.g., recursive methods).

—Modules are extensible and use late binding. Unlike mainstream languages, late binding applies to both type and object members. It is possible to define both virtual methods and *virtual types* or *virtual classes* [Madsen and Moller-Pedersen 1989; Ernst et al. 2006].

—Module refinement resembles object-oriented inheritance; there is a subtype relationship defined between modules.

—Modules can be higher-order: they can be parameterized by other modules.

—It is possible to define mixin modules as functions that transform modules.

The operational semantics of DEEP is simple, but the type system (in particular the metatheory) is quite complex. There are three aspects of the type system that deserve special mention.

First, type and class definitions within a module can be virtual, which means that they can be overridden by derived modules. Virtual classes are not a new idea; they first appeared almost 20 years ago in the BETA language [Madsen and Moller-Pedersen 1989]. However, designing a type system that can handle the simultaneous refinement of a system of mutually recursive classes has proved to be very difficult, and it was not formalized until recently [Ernst et al. 2006].

Second, DEEP supports not only virtual types, but also higher-order subtyping [Steffen and Pierce 1994] with bounded quantification over such types [Compagnoni and Goguen 2003]. Higher-order subtyping is crucial to our treatment of feature composition, but it interacts with recursion and virtual types in subtle ways [Hutchins 2009]. To our knowledge, DEEP is the only calculus that is capable of handling this combination.

Third, subtyping is defined directly over modules, rather than classes or types. In fact, the DEEP calculus does not even have a typing relation; the type system is based entirely on subtyping. This is perhaps the most controversial part of the calculus, but it simplifies the theory in various ways [Hutchins 2009].

The operational semantics of *g*DEEP, which defines how features are composed, is defined below. The type theory is presented in Appendix A.

4.1 *g*DEEP Syntax and Semantics

The *g*DEEP module system is a generalization of the DEEP module system and thus has all of the properties mentioned above. *g*DEEP provides a module system that supports feature composition. It does not handle the syntax, evaluation, or typing of expressions in the target artifact language (Java, Bali, etc.); these must be provided by the "sister calculus" that it is paired with.

Figure 4 shows the syntax and the operational semantics of *g*DEEP. The syntax is divided into two parts. The first part, shown on the left, is a calculus for features. This part represents the core calculus, and it is the only part which is common across all applications of *g*DEEP.

The second part of the calculus, shown on the right, is *artifact-specific*. These terms serve as placeholders for the particular language to which *g*DEEP 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 (see Section 4.8).

4.2 Notation

We use the following notational conventions.

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:
Top	empty module
X	variable
$\lambda^+ X \leq M.\ N$	monotone function
μX refines $M\ \{\overline{D}\}[\xi]$	module
$M(N)$	function application
$M@(N).L$	delegation
V, W ::=	Values:
$\lambda^+ X \leq M.\ N$	function
μX refines Top $\{\overline{D}\}[\xi]$	module

D, E ::=	Declarations:
$L : M$	nested module
$l : d$	artifact declaration
override $L : M$	module refinement
override $l : d$	artifact refinement

Artifact-specific constructs:

d, e ::=	Artifact-specific decl.:
$M@(N).l$	delegation
...	(unspecified)
ξ ::=	Artifact-specific info:
$\bullet$	empty
...	(unspecified)

Evaluation context: $E ::= [] \mid E(M) \mid V(E) \mid E@(M).L \mid V@(E).L \mid \mu X$ refines $E\ \{\overline{D}\}$

$$\frac{M \longrightarrow M'}{E[M] \longrightarrow E[M']} \qquad (\lambda^+ X \leq M.\ N)(V) \longrightarrow [X \mapsto V]N \quad \begin{matrix}\text{(E-Cong)}\\ \text{(E-App)}\end{matrix}$$

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

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

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

$$\frac{\overline{C} \uplus \overline{D} \neq \overline{E}}{\mu X \text{ refines } (\mu X \text{ refines Top } \{\overline{C}\}[\xi_1])\ \{\overline{D}\}[\xi_2] \longrightarrow \text{error}} \quad \text{(E-Error)}$$

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

$$\begin{matrix} L : M \uplus \text{override } L : N = L : N \\ l : d \quad \uplus \text{override } l : e \quad = l : e \end{matrix} \quad \text{(D-Comp)}$$

$$\bullet \uplus \bullet = \bullet \quad (\xi\text{-Comp})$$

Fig. 4. Syntax and operational semantics of gDeep.

—$\overline{D}$ 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 labeled L or l, resp., 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 the term M for the variable X within N.

—$M.L$ and $M.l$ are syntactic sugar for $M@(M).L$ and $M@(M).l$, respectively.
—$\mathsf{original}_X.\ell$, is syntactic sugar for $M@(X).\ell$, when the original keyword appears in the context: μX refines M {...$\mathsf{original}_X.\ell$...}. Furthermore, the X may be omitted, for example, original.ℓ, in cases that are unambiguous.

4.3 Modules

A module is declared using the syntax μX refines M $\{\overline{D}\}[\xi]$. The variable X provides a name for "self" within the module, much like the keyword this in Java. Modules in *g*DEEP can be nested, so it is important that each module has a unique name for "self". The self-variable allows a declaration within a module to refer to other declarations within the same module using a *path* that starts with X, for example, $X.L$.

$\overline{D}$ is a sequence of zero or more declarations, in which a declaration is a labeled (i.e., named) term. The set ℓ of labels is divided into the two subsets L and l. A declaration $L : N$ is a compound declaration, which is an ordinary term N in *g*DEEP. A declaration $l : d$ is an atomic declaration, which is a term d in the artifact language. Since *g*DEEP is defined independently of any particular artifact language, atomic modules are treated as raw "chunks of code" and are not otherwise interpreted.

The refines M clause denotes the parent of the module, which is much like a superclass in Java. A module that has no parent can declare Top to be the parent. A module extends its parent by adding new declarations or by overriding existing declarations. Declarations that override existing ones must be declared with keyword override.

ξ is an artifact-specific annotation. It can be used by the artifact language for the composition of artifact-specific details, but is otherwise ignored by the core calculus (see Section 4.8).

4.4 Inheritance and Composition

The following example demonstrates how composition works in *g*DEEP. For the purpose of discussion, we will use a very simple artifact language in which declarations $l : d$ have the form l : Int $= t$, in which t is a simple arithmetic expression.

```
A = μX refines Top { a: Int = 1; b: Int = 3; };
B = μX refines A { override a: Int = 2; c: Int = 4; };
```

In this example, A is a base module, and B inherits from (i.e., refines) A. Note the use of keyword override. Overriding declarations must be declared as such; it is an error to have two declarations with the same name.

In *g*DEEP, refinement is a computation that is performed at composition time. Evaluating B will merge the definitions in B with the definitions in A. Overriding definitions replace those in their parent:

```
B —» μX refines Top { a: Int = 2; b: Int = 3; c: Int = 4; };
```

4.5 Paths and Delegation

Delegation, written using the syntax $M@(N).\ell$, projects the declaration named ℓ from the module 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. The body of a declaration may refer to "self" using the self-variable of the module. In most object-oriented languages, including Java, "self" is treated as a hidden argument, which is passed implicitly during a method call. In *g*DEEP, the "self" argument is not hidden; it is passed explicitly.

Usually, M and N are the same term. The standard object-oriented dot notation $M.\ell$ is syntax sugar for $M@(M).\ell$. In other words, the expression $M.\ell$ projects the slot named ℓ from M, passing M as "self." The case in which M and N are different arises when a derived module wishes to delegate behavior to its parent, as will be discussed shortly.

The following example demonstrates how delegation works in the simple case: expressions of the form $M.\ell$.

```
M1 = μX {
  A: μY refines Top { a: Int = 1; };
  B: μZ refines X.A { b: Int = 2; };
};
```

In this example, M1 is defined as a module that contains two nested modules: A and B. A is the base module, whereas B inherits (i.e., refines) from A. Note that B refers to A using the path $X.A$ – it projects A from the self-variable X.

Like object-oriented languages, *g*DEEP uses *late binding*. Variable X is not assigned a value until B is actually projected from M1, for example,

```
M1.B ⟶ μY refines M1.A { b: Int = 2; } ⟶» μY refines Top { a: Int = 1; b: Int = 2; }
```

In the next example, we create a new module M2 that refines A within M1 by overriding declaration a. Note, the new definition of A refines the old definition by inheriting from it.

```
M2 = μX refines M1 {
  // original_X.A is syntactic sugar for M1@(X).A
  override A: μY refines original_X.A { override a: Int = 2; };
};
```

Because of late binding, refining the definition of A will automatically affect the definition of B. M2 inherits B from M1. However, when the expression M2.B is evaluated, X will be bound to M2 rather than M1, for example,

```
M2.B ⟶ μY refines M2.A { b: Int = 2; } ⟶
    μY refines (μY refines M1@(M2).A { override a: Int = 2; }) { b: Int = 2; } ⟶»
    μY refines Top { a: Int = 2; b: Int = 2; }
```

This example also illustrates a more complex use of delegation. Notice that the module A is overridden with a version that inherits from original_X.A. The "original" module in this case is M1, so original_X.A is syntax sugar for M1@(X).A. The expression M1@(X).A means: "extract the declaration named A from M1, but pass X, which is the self-variable for M2, as self."

Delegation is similar to the use of keyword super in Java or Super in Jak. When a derived module inherits from a base module, it can delegate some behavior to its parent. Delegation allows a feature to "refine" its parent by transforming existing declarations, rather than just overriding these declarations outright [Batory et al. 2004; Apel et al. 2008b]. Note that although we refer to this mechanism as "delegation", it is actually done statically, when features are composed.

Late binding allows a base module to defer implementation details to derived modules. All references to "self" within the base module are mapped to the derived module when features are composed. Delegation, in turn, allows a derived module to refer back to the base module. Module inheritance thus establishes a two-way communications link between parent and child.

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, because $N = M$. It is also safe for a derived module to delegate to its parent.

In addition, every overriding declaration in a derived module must be a subtype of the original declaration in its parent. In the example above, note that the definition of A within M2 inherits from $\mathsf{original}_X$.A. This pattern of inheritance is enforced by *g*DEEP's static type system.

4.6 Monotone Functions

In *g*DEEP, base features are represented as modules, whereas refinements are represented as functions over modules. The expression $\lambda^+X \leq M.\ N$ is a function that accepts any subtype of M as an argument. Function application is written as $M(N)$.

The subtype relation extends to functions as well as modules. Subtyping between functions is point-wise; given two functions F and G:

$F \leq G$ if and only if $F(A) \leq G(A)$ for all A.

Unlike most other module calculi, functions in *g*DEEP are *monotone*, which is why they are declared with the curious λ^+ notation. Monotone functions have an additional property: if F is a monotone function then:

$A \leq B$ implies $F(A) \leq F(B)$

We provide only monotone functions in *g*DEEP because we are interested in encoding features. The full DEEP calculus includes general-purpose (i.e., nonmonotone) functions as well, but such functions are not needed for FOP, so we have omitted them in the interest of simplicity.

A feature which can be applied to a base program of type A is written as a function that refines an argument of type A, for example,

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

This feature takes an argument X and extends it by adding the declarations given in {...}.

Type Constraints. Functions in DEEP have an additional restriction because they are monotone. Given a function $\lambda^+X \leq M.\ N$, the variable X can appear

only in *covariant* positions within N, which means that it cannot appear within the argument type of functions.

The subtyping laws shown here are drawn from theory of *polarized higher-order subtyping* for System $F^{\omega}_{\leq}$, as developed by Steffen and Pierce [Steffen and Pierce 1994; Steffen 1997]. *g*DEEP extends this theory by using *bounded quantification* [Cardelli and Wegner 1985; Compagnoni and Goguen 2003] exclusively to establish type constraints on formal arguments. The application $M(N)$ is valid only if N is a subtype of the argument type of M. Argument types in *g*DEEP are invariant rather than contravariant in subtypes; this avoids the well-known problem with contravariance [Pierce 1994].

4.7 Subtyping Laws for Feature Composition

A feature encapsulates a slice of program behavior. Applying a feature to a program extends 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 programs, then:

(1) $F(A) \leq A$ for any legal argument A

(2) $A \leq B$ implies $F(A) \leq F(B)$ for any legal arguments A, 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 identity function on modules. The second law states that, if a feature is applied to a more specific program, it will always generate a more specific result. The second law is guaranteed by the monotonicity requirement on functions.

Together, these two laws allow us to derive subtyping rules for feature compositions. If $F_1..F_n$ is a sequence of features, and $G_1..G_m$ is a sequence of features, then

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

if $F_1..F_n$ contains all of the features in $G_1..G_m$ *and all of these features are applied in the same order*. That is, $G_1..G_m$ must be a subsequence of $F_1..F_n$. For example, the feature composition $Mult(Eval(Expr))$ is a subtype of the composition $Mult(Expr)$, but it is not a subtype of $Eval(Mult(Expr))$ or $Var(Mult(Eval(Expr)))$.

4.8 Artifact-Specific Constructs

*g*DEEP provides two main "hooks" for integrating an artifact language. First, declarations in the artifact language are denoted by d (cf. Figure 4). In the case of Java, d represents field and method declarations; in the case of XML, d represents XML elements, and so on. Artifact declarations are completely opaque; *g*DEEP does not interpret them.

Second, compound modules of the artifact language such as Java classes or Xak modules are represented as compound modules in *g*DEEP. *g*DEEP's modules can be annotated with a domain-specific construct, named ξ. ξ is defined to hold any information about a compound module that is required by the artifact language, but that *g*DEEP does not include. For example, in our encoding of Java [Apel and Hutchins 2007], we use it to hold the class constructor along

with the extends clause (see Section 6). Using this information, it is possible to create a one-to-one mapping between Java classes and *g*DEEP modules. In Section 6 and Section 7, we explain in detail how Java, Bali, XML and Haskell are plugged into *g*DEEP.

5. TYPE CHECKING

We illustrate the role of *g*DEEP's type system informally by example; the type system of *g*DEEP is defined formally in Appendix A. The type system was designed to support separate compilation and type checking of features, something that existing tools such as Jak cannot do.

Type errors will be detected before composition. Examples of type errors are: referring to a slot (or calling a method) that does not exist, calling a function with an invalid argument, or overriding a declaration with a value that is not a subtype of the original. The following example illustrates these errors, using a simple artifact language that supports integer arithmetic and strings:

```
A = μ X { a: Int = 1; }
B = μ X refines A { b: Int = X.a + 1; } // ok
C = μ X { c: Int = X.a + 1; } // error
D = μ X refines A { override a: String = "1"; } // error
F = λ+ X ≤ A. μ Y refines X { ... }
G = F(B) // ok
H = F(C) // error
```

The definition of C is an error because the slot a does not exist. D attempts to override a with new definition that has a different type, and H calls F with an invalid argument.

5.1 Integrating an Artifact Type System

The *g*DEEP type system can check only *g*DEEP expressions; it must be paired with an artifact type system to check the artifact expressions. As illustrated in the previous example, however, the artifact language and *g*DEEP are intertwined. In particular, the artifact language may use paths such as $X.a$ (syntactic sugar for $X@(X).a$) in artifact declarations. Paths are the primary interface between the artifact language and *g*DEEP; they allow artifact code to traverse the *g*DEEP module structure. All of the languages that we consider in this paper rely on paths.

Figure 5 formally defines the simple artifact language that we have been using so far in our examples. The language has two types, Int and String, and supports arithmetic expressions and string concatenation. We have deliberately made the language very simple, but it nevertheless demonstrates the key mechanisms by which an artifact language can be integrated into *g*DEEP in a type-safe way.

The artifact language includes three rules that hook it to *g*DEEP. First, artifact reduction ($\longrightarrow_{\text{art}}$) must handle expressions of the form $M@(N).l$. It does so by invoking *g*DEEP reduction ($\longrightarrow$).

Second, artifact typing must assign a type to expressions of the form $M@(N).l$. It does so by invoking the *g*DEEP type system. It (1) ensures that $M@(N).l$ is well formed and (2) uses the *g*DEEP subtyping judgment to look up the declaration associated with $M@(N).l$.

$$
\begin{array}{lll}
n ::= & & \text{integer literals} \\
s ::= & & \text{string literals} \\
T ::= \mathsf{Int} \mid \mathsf{String} & & \text{artifact types} \\
t ::= n \mid s \mid M@(N).l \mid t + t & & \text{artifact expressions} \\
d ::= T = t & & \text{artifact declaration}
\end{array}
$$

$$
\begin{array}{ll}
n_1 + n_2 \longrightarrow_{\text{art}} n_3 & \text{where } n_3 \text{ is the sum of } n_1 \text{ and } n_2 \\
s_1 + s_2 \longrightarrow_{\text{art}} s_3 & \text{where } s_3 \text{ is the concatenation of } s_1 \text{ and } s_2 \\
M@(N).l \longrightarrow_{\text{art}} t & \text{if } M@(N).l \longrightarrow (T = t)
\end{array}
$$

$$
\frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \frac{}{\Gamma \vdash s : \mathsf{String}} \qquad \frac{\Gamma \vdash t_1 : \mathsf{T}, \quad t_2 : \mathsf{T}}{\Gamma \vdash t_1 + t_2 : \mathsf{T}}
$$

$$
\frac{\Gamma \vdash M@(N).l \text{ wf} \quad \Gamma \vdash M@(N).l \leq (T = t)}{\Gamma \vdash M@(N).l : T}
$$

$$
\frac{\Gamma \vdash t : T}{\Gamma \vdash T = t \text{ wf}} \qquad \frac{}{\Gamma \vdash (T = t_1) \leq (T = t_2)}
$$

Fig. 5. A simple artifact language.

Third, the artifact language must define subtyping and well-formedness rules for artifact declarations. These rules plug into the *g*DEEP type system; a module is only well formed if all of its declarations are well formed. In this case, an artifact declaration is well formed if it is well typed and if the type of the expression matches the type that was declared for it. One declaration is a subtype of another (i.e., one declaration can be overridden with another) if they have the same declared type.

The artifact language in Figure 5 is sound only if the type of *M*@(*N*).*l* is preserved under artifact reduction. Artifact reduction merely invokes *g*DEEP reduction, so this result follows from the soundness of *g*DEEP.

In addition, we would like to know that types in the artifact language are preserved by feature composition, that is, by *g*DEEP reduction. There are two *g*DEEP reduction rules. β-reduction substitutes one term for another and module composition overrides one slot with another. The soundness result for *g*DEEP guarantees that subtyping and well-formedness are preserved under substitution, and the subtyping rule for artifact declarations ensures that overriding declarations have the same type. Thus, feature composition preserves artifact types.

It is not possible to prove that *g*DEEP can be combined with any artifact language in a way that is sound. In most cases, however, the *g*DEEP type system and the artifact type system are largely orthogonal. The type safety result for *g*DEEP provides some basic guarantees that can be reused with many different artifact languages to prove that features are safe for use in those languages.

5.2 Link Errors

Not all errors can be detected before feature composition. In particular, there are two kinds of error that must be detected after composition: name clashes and missing implementations.

A name clash occurs when a derived module attempts to add a new declaration ℓ, but a declaration named ℓ already exists in the parent. For example:

```
A = μ X { a: Int = 1; }
B = μ X refines A { b: Int = 2; }
F = λ+ X ≤ A. μ Y refines X { b: String = "hello"; }
D = F(B)
```

All of the features above are well formed. The feature F can be applied to any subtype of A, and B is a subtype of A, so F(B) is well formed. However, evaluating F(B) will cause a name clash, because both F and B have declarations named b.

Missing implementations arise when *g*DEEP is paired with an artifact language that allows program elements to be declared without an implementation. The canonical example is an abstract method in Java; such a method has a type signature but no implementation.

Figure 1 shows an example of missing implementations. Assume feature *Expr* is implemented as a module and *Eval* and *Mult* are defined as functions that extend their input. Then *Eval*(*Mult*(*Expr*)) has a missing implementation—class Mult does not implement the eval method.

Both of these errors correspond exactly to link errors in other programming languages. For example, source files in C are type checked and compiled separately. The compilation phase guarantees that every routine is declared before it is used and the use of a routine matches the type that was declared for it. However, it not possible to determine, at compile-time, that every declared routine is actually implemented, because the compiler does not have access to all source files. Indeed, the whole point of separate compilation is that the compiler should not require access to all source files. As a result, inter-file dependencies are resolved by the linker, rather than by the compiler. The C linker will generate an error if there are any missing implementations or if there are two implementations with the same name (i.e., a name clash).

*g*DEEP uses a linking mechanism that is similar to C, except that it supports a hierarchical namespace based on features. Evaluating a *g*DEEP expression will compose features together. If composition generates a name clash, then the result will be an error, as defined by rule (E-ERROR).

Missing implementations are not automatically detected by *g*DEEP, because missing implementations are artifact-specific (e.g., missing implementations cannot occur in the simple artifact language shown in Figure 5). Instead, missing implementations will be detected after feature composition, when the artifact program is compiled.

To summarize, feature composition proceeds in three distinct phases:

(1) Static type checking before composition (*g*DEEP well-formedness).
(2) Composition of features (*g*DEEP evaluation—will detect name clashes).
(3) Compilation of artifact code (will detect missing implementations).

Static type checking ensures that classes, objects, methods, etc. are properly declared and used in a way that is consistent with their declaration. Linking

(phases 2 and 3) ensures that every declaration is implemented and that implementations do not conflict.

5.3 A Note on Name Clashes

Some other formalisms for mixin composition do not define name clashes as errors. For example, the semantics given by Flatt et al. [1998] associate each object with a view that defines which methods are visible. An object may have multiple methods with the same name, and the view determines which one is chosen.

There are several reasons why we have not pursued this approach. The most important reason is that the purpose of *g*DEEP is to provide a model that is consistent with existing feature composition tools such as Jak, Xak, and Bali. In these tools, the end result of feature composition is a program in the original artifact language that does not contain features. For example, Jak produces straight Java code that can be handed off to a standard Java compiler. Views are not a part of standard Java, so our semantics must treat name clashes as errors.

In addition, views are semantically complex, and there are easier ways to achieve a similar effect. Our preferred way is to use name mangling, which is commonly used in other mixin-based languages such as gbeta [Ernst et al. 2006]. Each feature is assigned a unique name, and each non-overriding slot is renamed to one that incorporates the name of its container. Overriding slots are declared as such and are renamed to the name of the declaration that they override. Ambiguous overrides and ambiguous paths are flagged as errors during the name mangling phase, which precedes the type checking phase.

Name mangling is good enough to prevent accidental name clashes, such as the "artistic cowboy" that inherits a draw method from both Artist and Gunslinger. In this case, the methods would have been renamed draw_Artist and draw_Gunslinger. Name mangling does not handle the case where a composition includes the same feature multiple times (i.e., an artist who draws with both oil paint and water color) but this situation does not typically arise in FOP [Apel et al. 2008c, 2010b] and in the code factorization problems that tools like AHEAD were designed to handle.

6. INTEGRATING JAVA WITH *g*DEEP

In order to illustrate the capabilities and the generality of *g*DEEP, we begin with a complex example: the integration of Java with *g*DEEP.

To integrate Java with *g*DEEP, we must define a bijective (i.e., one-to-one and onto) mapping between Java programs and *g*DEEP modules. Our basic strategy will use this mapping to translate Java features to *g*DEEP, use *g*DEEP to perform feature composition, and then translate the resulting program back to Java. We will write the mapping function, which translates Java to *g*DEEP, as $\langle\!\langle \_ \rangle\!\rangle$.

Member variables, constructors, and method declarations in Java are translated verbatim into atomic declarations in *g*DEEP. For member variables, we use the name of the variable as the name of the declaration. For methods

and constructors, we use a name-mangling scheme to ensure that overloaded methods are assigned unique names, for example,

⟪ int x; ⟫ = x: (int x);
⟪ void m(int x) { ... } ⟫ = m1_i: (void m(int x) { ... });
⟪ A() { ... } ⟫ = new0: A() { ... }

Classes and interfaces are translated into modules in *g*DEEP. Since *g*DEEP defines its own notion of inheritance, we could attempt to map Java inheritance onto *g*DEEP inheritance. There are two problems with this approach. First, we would like features to be a minimally invasive extension. Replacing Java inheritance with *g*DEEP inheritance would significantly alter the structure of the language.

Second, the inheritance requirements imposed by *g*DEEP are slightly different than those imposed by Java. In particular, constructors in *g*DEEP are treated like virtual methods; they are inherited by derived modules. Constructors *must* be inherited in order for class refinement to work properly; if the refinement of a class was to alter constructor signatures, it would break code in other features that attempted to instantiate the class.

Instead of trying to map Java inheritance onto *g*DEEP inheritance, we simply encode the extends and implements clauses as artifact-specific annotations,

⟪ class B extends A { ... } ⟫ = B: μY refines Top { ⟪ ... ⟫ } [class B extends X.A]

The final part of the translation is to map global class names like Bar onto paths of the form X.Bar. In Java, all class names are treated as global identifiers. In FOP, classes are not global—a class definition is local to the feature in which it is defined. The translation from global names to local paths ensures that classes are virtual and use late binding, which is the key to making feature composition work. Figure 6 shows an excerpt of the earlier Java/Jak example encoded in *g*DEEP

6.1 Modular Type Checking

Type checking in Java is largely unaffected by the translation into *g*DEEP, with one exception: the fact that class names are translated into local paths.

Formal models of Java type checking, such as Featherweight Java [Igarashi et al. 1999] or ClassicJava [Flatt et al. 1998], make use of a global class lookup table. Type judgments then use this table to find the type signatures of methods and constructors.

We perform modular type checking within a feature by using *g*DEEP to perform class lookups. We replace class lookups of the form:

CT(B) = class B extends A { ... }

where CT is the global class table, with *g*DEEP subtyping judgments of the form:

X.B $\leq$ ⟪ class B extends A { ... } ⟫

All other typing rules are identical to ordinary Java. We now consider whether a Java program that is well typed before feature composition will remain well typed after composition. Feature composition is performed by *g*DEEP

Feature *Expr*

```
1  ExprFeature = μX refines Top {
2    Expr: μY refines Top {
3      toString0: abstract String toString();
4    } [ (abstract class Expr) ];
5
6    Val: μY refines Top {
7      val: int val;
8      new1_i: Val(int n) { val = n; }
9      override toString0: String toString() { return String.valueOf(val); };
10   } [ (class Val extends X.Expr) ];
11
12   Add: μY refines Top {
13     a: X.Expr a;
14     b: X.Expr b;
15     new2_ii: Add(X.Expr e1, X.Expr e2) { a = e1; b = e2; }
16     override toString0: String toString() { return a.toString() + "+" + b.toString(); };
17   } [ (class Add extends X.Expr) ];
18 };
```

Feature *Eval* refines *Expr*

```
19 EvalFeature = λ+Z ≤ ExprFeature. μX refines Z {
20   override Expr: μY refines original_X.Expr {
21     eval0: abstract int eval();
22   } [ ];
23   override Val: μY refines original_X.Val {
24     override eval0: int eval() { return val; };
25   } [ ];
26   override Add: μY refines original_X.Add {
27     override eval0: int eval() { return a.eval() + b.eval(); };
28   } [ ];
29 };
```

Feature *Mult* refines *Expr*

```
30 MultFeature = λ+Z ≤ ExprFeature. μX refines Z {
31   Mult: μY refines Top {
32     a: X.Expr a;
33     b: X.Expr b;
34     new2_ii: Mult(X.Expr e1, X.Expr e2) { a = e1; b = e2; }
35     override toString0: String toString() { return "(" + a.toString() + "*" + b.toString() + ")"; }
36   } [ (class Mult extends X.Expr) ];
37
38   override Add: μY refines original_X.Add {
39     override toString0: String toString() { return "(" + original_Y.toString() + ")"; };
40   };
41 }
```

Fig. 6. The "expression problem" encoded in *g*DEEP.

reduction, so we must show that typing derivations in Java are preserved under reduction in *g*DEEP. Typing derivations make use of *g*DEEP subtyping and well-formedness, so as with the simple artifact language shown in Section 5.1, this result follows from the soundness of *g*DEEP [Apel and Hutchins 2007].

There is one remaining subtlety. Notice that we have replaced an equality in type lookup with an inequality. Java type checking uses the fact that class lookup is an equality to determine whether methods are overriding or not. This situation is very similar to the name clashes discussed in Section 5.2, and we can resolve it in the same way. We recognize the fact that composition may introduce a name clash and signal an error if it does.

Feature *Expr*

```
1  ExprFeature = μX refines Top {
2    Expr: X.Val | X.Expr X.Oper X.Expr;
3    Oper: '+';
4    Val : INTEGER;
5  };
```

Feature *Mult* refines *Expr*

```
6  MultFeature = μX refines ExprFeature {
7    override Oper: original_X.Oper | '*';
8  };
```

Feature *Var* refines *Expr*

```
9  VarFeature = μX refines ExprFeature {
10   override Expr: original_X.Expr | X.Var | 'let' X.Var '=' X.Expr 'in' X.Expr;
11   Var : IDENTIFIER;
12 };
```

Fig. 7. The Bali example of Figure 2, written in *g*DEEP.

7. INTEGRATING FURTHER ARTIFACT LANGUAGES

We now illustrate how further artifact languages can be used with *g*DEEP. We begin with the Bali and XHTML examples from Section 2 and proceed with a further example written in Haskell. For Bali and XML, we have developed formalizations, called *g*BALI and *g*XAK, that we have integrated with the *g*DEEP calculus. A comprehensive description of the syntax and evaluation rules of *g*BALI and *g*XAK can be found elsewhere [Apel and Hutchins 2007].

7.1 Bali

A Bali grammar contains a set of production rules. Bali does not define its own compound modules; each grammar consists of a single top-level module, without any nested modules. Because Bali does not have nested modules, there are no Bali-specific module annotations and thus there is no need for a translation function. *g*DEEP declarations and modules can be used as-is.

Rules in Bali can refer to other rules in the same grammar by name. In *g*BALI, such references are expressed as paths of the form $X.l$. They can also override rules of a base grammar and refer to the original definitions via $\mathsf{original}_X.l$. Both of these terms are syntax sugar for standard *g*DEEP delegation—$M@(N).l$. Figure 7 shows how the earlier Bali example is encoded in *g*DEEP.

7.2 XML

An XML document contains a set of elements, which are organized in a hierarchical structure. One might assume that, because XML documents already have a named hierarchical structure, we might attempt to define a translation function that maps XML elements onto *g*DEEP modules, in much the same way as we did for classes in Java. However, such a mapping is inappropriate for XML.

The names of declarations in a *g*DEEP program are used to establish a module structure. In contrast, the tags of an XML document are just markup—they may or may not have anything to do with modular structure. Moreover, names

Feature *Expr*

```
1  ExprFeature = μX refines Top {
2    Contents: ();
3    Operations: (<li>Addition of integers</li>);
4    Main: (<html>
5              <head><title>A Simple Expression Evaluator</title></head>
6              <body bgcolor="white">
7                <h1>A Simple Expression Evaluator</h1>
8                <?gdeep expr="X.Contents"?>
9                <h2>Supported Operations</h2>
10               <ul><?gdeep expr="X.Operations"?></ul>
11             </body>
12           </html>);
13 };
```

Feature *Eval* refines *Expr*

```
14 EvalFeature = μX refines ExprFeature {
15   override Contents:
16     (<?gdeep expr="original_X.Contents"?><h2>Evaluation of Arithmetic Expressions</h2>);
17 };
```

Feature *Mult* refines *Expr*

```
18 MultFeature = μX refines ExprFeature {
19   override Operations:
20     (<?gdeep expr="original_X.Operations"?><li>Multiplication of integers</li>);
21 };
```

Fig. 8. The Xak example of Figure 3, written in *g*DEEP.

of XML elements do not need to be unique in the scope of an enclosing tag, for example, as in the case of XHTML. *g*DEEP, on the other hand, requires that names are unique.

Xak explicitly establishes a module structure for XML by associating a module name with a particular XML element using the attribute xak:module. We use the same strategy for *g*XAK. Figure 8 shows an except of the previous XML code, written in *g*DEEP. This example divides the XML document explicitly into three named submodules: Contents, Operations, and Main. Note that, unlike the original definitions, Contents and Operations are no longer subelements of Main. Instead, Main refers to Contents and Operations by means of the paths X.Contents and X.Operations. *g*XAK differs from Xak because it uses paths rather than element annotations to establish a module structure. However, the net effect is the same.

7.3 Haskell

Figure 9 shows a simple implementation of the expression evaluator, supporting addition, evaluation, and multiplication, written Haskell and *g*DEEP. Haskell supports a style of programming that is not found in Java; data types and functions over those types are defined by means of cases.

As before, we must define a one-to-one function that maps Haskell code into *g*DEEP declarations. Our mapping for Haskell translates both data types and function definitions into atomic declarations. We represent all of the cases of a function by a single cases clause (Lines 4 and 9 on the right), which can reference previously defined cases by means of the original keyword.

Feature *Expr*

```
1  data Expr = Val Int | Add Expr Expr;
2
3
```

Feature *Eval* refines *Expr*

```
4  eval :: Expr −> Int;
5  eval (Val i) = i;
6  eval (Add a b) = (eval a) + (eval b);
7
```

Feature *Mult* refines *Expr*

```
8  data Expr = Super | Mult Expr Expr;
9
10
11
```

Derivative *Mult* + *Eval*

```
12 eval (Mult a b) = (eval a) ∗ (eval b);
13
14
15
```

Feature *Expr*

```
1  ExprFeature = μX refines Top {
2    Expr: data Expr = Val Int | Add X.Expr X.Expr;
3  };
```

Feature *Eval* refines *Expr*

```
4  EvalFeature = μX refines ExprFeature {
5    eval: function eval :: X.Expr −> Int =
6      cases (Val i) = i | (Add a b) = (X.eval a) + (X.eval b);
7  };
```

Feature *Mult* refines *Expr*

```
8  MultFeature = μX refines ExprFeature {
9    override Expr: data Expr =
10     original_X.Expr | Mult X.Expr X.Expr;
11 };
```

Derivative *Mult* + *Eval*

```
12 MultEvalFeature = μX refines ExprFeature {
13   override eval: function eval :: X.Expr −> Int =
14     cases original_X.eval | (Mult a b) = (X.eval a) ∗ (X.eval b);
15 };
```

Fig. 9. An expression evaluator in Haskell (left) and its encoding in *g*DEEP (right).

As with our Java translation, Haskell typing judgments are preserved by feature composition. A more detailed encoding that demonstrates type safety for variant data types can be found in Hutchins [2009]. Practical experiences with Haskell and FOP are discussed elsewhere [Apel et al. 2009b].

8. RELATED WORK

*g*DEEP is inspired by DEEP [Hutchins 2006], which is a formal object calculus that implements virtual classes [Madsen and Moller-Pedersen 1989] in a type-safe manner. Several other calculi have been developed for virtual classes, for example, by Ernst et al. [2006], Clarke et al. [2007], and Odersky et al. [2003], which could have been alternative starting points for our work.

Modules in *g*DEEP are similar in some respects to formal models of objects, for example, by Abadi and Cardelli [1996] and Boudol [2004]. Boudol in particular shows how extensible objects, classes, and even mixins can be implemented by using *generating functions*—functions that take "self" as their first argument. Modules in *g*DEEP are essentially generating functions, which are applied using delegation: $M@(N).L$. The *g*DEEP type system, however, is completely different from that used by Boudol, because *g*DEEP relies exclusively on subtyping, whereas Boudol uses record types with row-variables.

A significant body of work has explored the concept of mixin composition, for example, Bracha and Cook [1990], Findler and Flatt [1998], Flatt et al. [1998], Bono et al. [1999], Ancona et al. [2003], and Kamina and Tamai [2004]. *g*DEEP builds on this work and implements deep mixin composition [Zenger and Odersky 2005] in a language-independent manner. McDirmid et al. [2006] have presented a formal system for modular linking in the presence of mixins. They aim similarly at language independence but with focus on object orientation.

It is an interesting issue how to generalize their formal system to noncode languages.

A concept related to mixins is the concept of a trait. A trait is a reusable unit of behavior that, in its original proposal, has no state [Ducasse et al. 2006]. Experiences from practical FOP show that features often have state, or to phrase it differently, add fields to existing classes. Although there is a recent proposal for stateful traits [Bergel et al. 2008], we favored the mixin concept because of it is widely used in FOP tools. Another difference between traits and mixins lies in the treatment of name clashes. Traits explicitly require the composer to handle conflicts. Anyway, this paper does not focus on these issues but on the principle of uniformity, which is also applicable to the work on traits.

Multimethods have been proposed to problems of single method dispatch [Millstein and Chambers 2002], for example, the binary method problem. Multiple method dispatch allows a programmer to extend a given object subsequently without changing existing code or introducing type errors, which is also possible with *g*DEEP. Additionally, *g*DEEP structures the name space hierarchically in order to encapsulate and scale the extensions a feature can apply, which has been shown useful in FOP [Smaragdakis and Batory 2002; Batory et al. 2004].

Several approaches aim at ensuring the correctness of feature composition. Apel et al. [2008a] extend Featherweight Java with constructs for feature composition, in particular, with constructs for the refinement of classes, constructors, and methods. It is not obvious how to generalize these results to arbitrary languages. Thaker et al. [2007], Delaware et al. [2009], Kästner and Apel [2008], and Apel et al. [2010a] use a combination of a SAT solver and an ordinary type system to check whether all program variants that can be composed from a set of features are type safe. Again, all of these approaches focus on Java only.

Li et al. [2002, 2005] proposed a technique to verify feature-oriented systems in two steps: (1) features are verified modularly and (2) their composition is verified without the need of re-verifying the involved features again. Their approach relies on three-valued model checking to take the "openness" of feature into account. The state machine models they use are quite spartan and do not consider noncode artifacts explicitly.

Features and feature composition can also be expressed in terms of algebra [Lopez Herrejon et al. 2006; Apel et al. 2008c, 2010b]. The advantage of an algebra-based approach is that reasoning about feature composition is simpler than in *g*DEEP. The disadvantage is that algebraic approaches operate at a higher level of abstraction. Algebraic approaches do not provide an operational semantics and type system, so it is harder to prove certain properties, and they cannot be easily used as a basis for implementations. We believe that both abstraction levels are equally important and useful for exploring the principles of feature composition.

We have developed a tool, called FEATUREHOUSE, that composes features written in Java, C#, C, Haskell, Bali, JavaCC, XML, Alloy following the rules of *g*DEEP, but a discussion of the rationales and details of the implementation

of FeatureHouse is outside the scope of this article and reported elsewhere [Apel and Lengauer 2008; Apel et al. 2009c, 2009a, 2009b]. We have used FeatureHouse in seven case studies of different sizes (2–99 features with about 300–90,000 lines of code) involving different types of artifacts (Java, C#, C, Haskell, JavaCC, XML). The case studies demonstrate the practicality of language-independent feature composition and reveals some insights in the mandatory and optional properties of language to be represented in *g*Deep. Details about the case studies are out of scope of this paper and are available on the Web.[2]

There are alternative composition models that relate to feature composition as modeled by *g*Deep. *Aspect-oriented programming* (*AOP*) aims at improving modularity of crosscutting concerns [Kiczales et al. 1997]. It has been observed that features are frequently crosscutting in nature [Mezini and Ostermann 2004; Apel et al. 2008b], and so it is not surprising that the techniques used in FOP are also associated with AOP [McDirmid and Hsieh 2003; Mezini and Ostermann 2004]. In this sense, *g*Deep models a subset of AOP [Apel et al. 2008b]. *Multidimensional separation of concerns* (*MDSC*) [Tarr et al. 1999] is very related to FOP [Apel et al. 2008b]. It favors the hierarchical nesting and composition of software artifacts much like FOP but allows programmers to define explicit composition rules for merging to pieces of software. Hence, *g*Deep is possibly a proper means for modeling MDSC and exploring the relationship between AHEAD and MDSC.

9. CONCLUSION

We have developed *g*Deep as a core calculus for feature-oriented programming (FOP) that encapsulates the essence of feature composition and validation. It abstracts from artifact-specific details and treats many different kinds of software artifact in a uniform way. *g*Deep provides three basic concepts for constructing features which are largely orthogonal: (1) a module allows a mutually recursive set of named definitions; (2) refinement statements allow a module to be extended; (3) monotone functions allow separate extensions to be composed.

We have presented the 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. We have demonstrated that a wide variety of very different artifact languages can be used with *g*Deep in order to enhance them with feature composition capabilities. In an accompanying technical report, we explain how we adapted and developed formalizations of Java, Bali, and XML and plugged them into *g*Deep [Apel and Hutchins 2007].

Our calculus serves also as an intermediate language for feature representation and manipulation, which is a foundation for large-scale feature-oriented program synthesis [Batory 2007]. Our tool FeatureHouse implements feature composition following the principles of *g*Deep. At the time of writing, FeatureHouse is able to compose feature written in Java, C, C#, Haskell, Bali, JavaCC, XML, and Alloy. Several case studies demonstrate the practicality and

[2]http://www.fosd.de/fh/.

scalability of our approach [Apel and Lengauer 2008; Apel et al. 2009c, 2009a, 2009b].

Beside composition, further algorithms and tools can be developed on top of the calculus to provide a seamless infrastructure for developing, analyzing, composing, and validating features in different representations.

ACKNOWLEDGMENTS

We thank Don Batory, Matthias Felleisen, Christian Kästner, Christian Lengauer, Marko Rosenmüller, and Yannis Smaragdakis for their comments.

REFERENCES

ABADI, M. AND CARDELLI, L. 1996. *A Theory of Objects*. Springer-Verlag, Berlin, Germany.

ANCONA, D., LAGORIO, G., AND ZUCCA, E. 2003. Jam—Designing a Java extension with mixins. *ACM Trans. Prog. Lang. Syst. 25*, 5, 641–712.

ANFURRUTIA, F., DÍAZ, O., AND TRUJILLO, S. 2007. On refining XML artifacts. In *Proceedings of International Conference on Web Engineering (ICWE)*. Lecture Notes in Computer Science, vol. 4607. Springer-Verlag, 473–478.

APEL, S. AND HUTCHINS, D. 2007. An overview of the gDeep calculus. Tech. rep. MIP-0712, University of Passau.

APEL, S., JANDA, F., TRUJILLO, S., AND KÄSTNER, C. 2009a. Model superimposition in software product lines. In *Proceedings of the International Conference on Model Transformation (ICMT)*. Lecture Notes in Computer Science, vol. 5563. Springer-Verlag, Berlin, Germany, 4–19.

APEL, S. AND KÄSTNER, C. 2009. An overview of feature-oriented software development. *Journal of Object Technology 8*, 5, 49–84.

APEL, S., KÄSTNER, C., GRÖSSLINGER, A., AND LENGAUER, C. 2009b. Feature (De)composition in functional programming. In *Proceedings of the International Conference on Software Composition (SC)*. Lecture Notes in Computer Science, vol. 5634. Springer-Verlag, Berlin, Germany, 9–26.

APEL, S., KÄSTNER, C., GRÖSSLINGER, A., AND LENGAUER, C. 2010a. Type safety for feature-oriented product lines. *Automat. Softw. Eng*.

APEL, S., KÄSTNER, C., AND LENGAUER, C. 2008a. Feature Featherweight Java: A calculus for feature-oriented programming and stepwise refinement. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, New York, 101–112.

APEL, S., KÄSTNER, C., AND LENGAUER, C. 2009c. FeatureHouse: Language-independent, automated software composition. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press, Los Alamitos, CA, 221–231.

APEL, S., LEICH, T., ROSENMÜLLER, M., AND SAAKE, G. 2005. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. Lecture Notes in Computer Science, vol. 3676. Springer-Verlag, Berlin, Germany, 125–140.

APEL, S., LEICH, T., AND SAAKE, G. 2008b. Aspectual feature modules. *IEEE Trans. Softw. Eng. 34*, 2, 162–180.

APEL, S. AND LENGAUER, C. 2008. Superimposition: A language-independent approach to software composition. In *Proceedings of the International Symposium on Software Composition (SC)*. Lecture Notes in Computer Science, vol. 4954. Springer-Verlag, Berlin, Germany, 20–35.

APEL, S., LENGAUER, C., MÖLLER, B., AND KÄSTNER, C. 2008c. An algebra for features and feature composition. In *Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST)*. Lecture Notes in Computer Science, vol. 5140. Springer-Verlag, Berlin, Germany, 36–50.

APEL, S., LENGAUER, C., MÖLLER, B., AND KÄSTNER, C. 2010b. An algebraic foundation for automatic feature-based program synthesis. *Sci. Comput. Prog.*.

BATORY, D. 2007. From implementation to theory in product synthesis. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*. ACM, New York, 135–136.

BATORY, D., SARVELA, J., AND RAUSCHMAYER, A. 2004. Scaling step-wise refinement. *IEEE Trans. Softw. Eng. 30*, 355–371.

BERGEL, A., DUCASSE, S., NIERSTRASZ, O., AND WUYTS, R. 2008. Stateful traits and their formalization. *Comput. Lang. Syst. Struct. 34*, 2–3, 83–108.

BONO, V., PATEL, A., AND SHMATIKOV, V. 1999. A core calculus of classes and mixins. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 1628. Springer-Verlag, Berlin, Germanym 43–66.

BOSCH, J. 1999. Super-imposition: A component adaptation technique. *Inf. Softw. Tech. 41*, 5, 257–273.

BOUDOL, G. 2004. The recursive record semantics of objects revisited. *J. Funct. Prog. 14*, 3, 263–315.

BOUGE, L. AND FRANCEZ, N. 1988. A compositional approach to superimposition. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*. ACM, New York, 240–249.

BRACHA, G. AND COOK, W. 1990. Mixin-based inheritance. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) and of the European Conference on Object-Oriented Programming (ECOOP)*. ACM, New York, 303–311.

CARDELLI, L. AND WEGNER, P. 1985. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv. 17*, 4, 471–522.

CLARKE, D., DROSSOPOULOU, S., NOBLE, J., AND WRIGSTAD, T. 2007. Tribe: A simple virtual class calculus. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*. ACM, New York, 121–134.

COMPAGNONI, A. AND GOGUEN, H. 2003. Typed operational semantics for higher order subtyping. *Inf. Computation 184*, 2, 242–297.

COOK, W. 1991. Object-oriented programming versus abstract data types. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*. Lecture Notes in Computer Science, vol. 489. Springer-Verlag, Berlin, Germany, 151–178.

DELAWARE, B., COOK, W., AND BATORY, D. 2009. Fitting the pieces together: A machine-checked model of safe composition. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*. ACM, New York, 243–252.

DUCASSE, S., NIERSTRASZ, O., SCHÄRLI, N., WUYTS, R., AND BLACK, A. 2006. Traits: A mechanism for fine-grained reuse. *ACM Trans. Prog. Lang. Syst. 28*, 2, 331–388.

ERNST, E. 2001. Family polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 2072. Springer-Verlag, Berlin, Germany, 303–326.

ERNST, E. 2003. Higher-order hierarchies. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 2743. Springer-Verlag, Berlin, Germany, 303–329.

ERNST, E., OSTERMANN, K., AND COOK, W. 2006. A virtual class calculus. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*. ACM, New York, 270–282.

FINDLER, R. AND FLATT, M. 1998. Modular object-oriented programming with units and mixins. In *Proceedings of the International Conference on Functional Programming (ICFP)*. ACM, New York, 94–104.

FLATT, M., KRISHNAMURTHI, S., AND FELLEISEN, M. 1998. Classes and mixins. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*. ACM, New York, 171–183.

HUTCHINS, D. 2006. Eliminating distinctions of class: Using prototypes to model virtual classes. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, New York, 1–19.

HUTCHINS, D. 2009. Pure Subtype Systems: A Type Theory For Extensible Software. Ph.D. dissertation, University of Edinburgh, Edinburgh, UK.

HUTCHINS, D. 2010. Pure Subtype Systems. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*. ACM, New York, 287–298.

IGARASHI, A., PIERCE, B., AND WADLER, P. 1999. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, New York, 132–146.

JOHNSSON, T. 1985. Lambda lifting: Transforming programs to recursive equations. In *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture (FPCA)*. Springer-Verlag, Berlin, Germany, 190–203.

KAMINA, T. AND TAMAI, T. 2004. McJava—A design and implementation of Java with mixin-types. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*. Lecture Notes in Computer Science, vol. 3302. Springer-Verlag, Berlin, Germany, 398–414.

KÄSTNER, C. AND APEL, S. 2008. Type-checking software product lines—A formal approach. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society Press, Los Alamitos, CA, 258–267.

KÄSTNER, C., APEL, S., UR RAHMAN, S., ROSENMÜLLER, M., BATORY, D., AND SAAKE, G. 2009. On the impact of the optional feature problem: Analysis and case studies. In *Proceedings of the International Software Product Line Conference (SPLC)*. Carnegie Mellon University, Pittsburgh, PA, 181–190.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 1241. Springer-Verlag, Berlin, Germany, 220–242.

KRISHNAMURTHI, S., FELLEISEN, M., AND FRIEDMAN, D. 1998. Synthesizing object-oriented and functional design to promote re-use. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 1445. Springer-Verlag, Berlin, Germany, 91–113.

LI, H., KRISHNAMURTHI, S., AND FISLER, K. 2002. Verifying cross-cutting features as open systems. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*. ACM, New York, 89–98.

LI, H., KRISHNAMURTHI, S., AND FISLER, K. 2005. Modular verification of open features using three-valued model checking. *Automat. Softw. Eng. 12*, 3, 349–382.

LIU, J., BATORY, D., AND LENGAUER, C. 2006. Feature-oriented refactoring of legacy applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, New York, 112–121.

LOPEZ-HERREJON, R., BATORY, D., AND LENGAUER, C. 2006. A disciplined approach to aspect composition. In *Proceedings of the International Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM, New York, 68–77.

MADSEN, O. AND MOLLER-PEDERSEN, B. 1989. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, New York, 397–406.

MCDIRMID, S. AND HSIEH, W. 2003. Aspect-oriented programming with Jiazzi. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*. ACM, New York, 70–79.

MCDIRMID, S., HSIEH, W., AND FLATT, M. 2006. A framework for modular linking in OO languages. In *Proceedings of the Joint Modular Languages Conference (JMLC)*. Lecture Notes in Computer Science, vol. 4228. Springer-Verlag, Berlin, Germany, 116–135.

MEZINI, M. AND OSTERMANN, K. 2004. Variability management with feature-oriented programming and aspects. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*. ACM, New York, 127–136.

MILLSTEIN, T. AND CHAMBERS, C. 2002. Modular statically typed multimethods. *Inf. Comput. 175*, 1, 76–118.

NYSTROM, N., CHONG, S., AND MYERS, A. 2004. Scalable extensibility via nested inheritance. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, New York, 99–115.

ODERSKY, M., CREMET, V., RÖCKL, C., AND ZENGER, M. 2003. A nominal theory of objects with dependent types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 2743. Springer-Verlag, Berlin, Germany, 201–224.

OSSHER, H. AND HARRISON, W. 1992. Combination of inheritance hierarchies. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, New York, 25–40.

PIERCE, B. 1994. Bounded quantification is undecidable. *Inf. Comput. 112*, 1, 131–165.

PIERCE, B. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA.

PREHOFER, C. 1997. Feature-oriented programming: A fresh look at objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 1241. Springer-Verlag, Berlin, Germany, 419–443.

REYNOLDS, J. 1994. User-defined types and procedural data structures as complementary approaches to data abstraction. In *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. MIT Press, Cambridge, MA, 13–23.

SMARAGDAKIS, Y. AND BATORY, D. 2002. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Method. 11*, 2, 215–255.

STEFFEN, M. 1997. Polarized higher-order subtyping. Ph.D. dissertation, University of Erlangen-Nuremberg, Nuremberg, Germany.

STEFFEN, M. AND PIERCE, B. 1994. Higher-order subtyping. Tech. rep. ECS-LFCS-94-280, University of Edinburgh, Edinburgh, UK.

TARR, P., OSSHER, H., HARRISON, W., AND SUTTON, JR., S. 1999. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press, Los Alamitos, CA, 107–119.

TERESE. 2003. Term rewriting systems. In *Cambridge Tracts in Theoretical Computer Science*. Vol. 55. Cambridge University Press.

THAKER, S., BATORY, D., KITCHIN, D., AND COOK, W. 2007. Safe composition of product lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, New York, 95–104.

TORGERSEN, M. 2004. The expression problem revisited. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 3086. Springer-Verlag, Berlin, Germany, 123–143.

TRUJILLO, S., BATORY, D., AND DIAZ, O. 2006. Feature refactoring a multi-representation program into a product line. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, New York, 191–200.

WRIGHT, A. AND FELLEISEN, M. 1994. A syntactic approach to type soundness. *Inf. Comput. 115*, 1, 38–94.

ZENGER, M. AND ODERSKY, M. 2005. Independently extensible solutions to the expression problem. In *Proceedings of the International Workshop on Foundations of Object-Oriented Languages (FOOL)*. ACM, New York.

Received March 2009; revised August 2009; accepted November 2009