Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement

Sven Apel
Dept. of Informatics and Mathematics
University of Passau, Germany
apel@uni-passau.de

Christian Kästner
School of Computer Science
University of Magdeburg, Germany
ckaestne@ovgu.de

Christian Lengauer
Dept. of Informatics and Mathematics
University of Passau, Germany
lengauer@uni-passau.de

Abstract

Feature-oriented programming (*FOP*) is a paradigm that incorporates programming language technology, program generation techniques, and stepwise refinement. In their GPCE'07 paper, Thaker et al. suggest the development of a type system for FOP to guarantee safe feature composition, i.e, to guarantee the absence of type errors during feature composition. We present such a type system along with a calculus for a simple feature-oriented, Java-like language, called *Feature Featherweight Java* (*FFJ*). Furthermore, we explore four extensions of FFJ and how they affect type soundness.

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

General Terms: Design, Languages, Theory

Keywords: Feature-Oriented Programming, Safe Feature Composition, Stepwise Refinement, Featherweight Java, Type Systems

1. Introduction

Feature-oriented programming (*FOP*) aims at the modularization of software systems in terms of features. A *feature* implements a stakeholder's requirement and is typically an increment in program functionality [40, 10]. Different variants of a software system are distinguished in terms of their individual features [26]. Contemporary feature-oriented programming languages and tools such as AHEAD [10], FSTComposer [9], and FeatureC++ [7] provide a variety of mechanisms that support the specification and composition of features properly. A key idea is that a feature, when added to a software system, introduces new structures, such as classes and methods, and refines existing ones, such as extending methods.

Stepwise refinement is a related software development paradigm that aligns well with FOP [45, 10]. In stepwise refinement, one adds detail to a program incrementally using refinements in order to satisfy a program specification. Refinements applied previously cannot affect the refinements applied subsequently, which is called henceforth the *principle of stepwise refinement*. In terms of FOP, the individual refinements implement features.

In prior work, features have been modeled as functions and feature composition as function composition [10, 31]. The function model tames feature composition in that it prevents features from affecting program structures that have been added by subsequent development steps, i.e., by features applied subsequently. This restriction is supposed to satisfy the principle of stepwise refinement. It decreases the potential interactions between different program parts (i.e., features) and avoids inadvertent interactions between present features and program elements that are being introduced in subsequent development steps [31, 35, 4].

In their GPCE'07 paper, Thaker et al. raised the question of how the correctness of feature-oriented programs can be checked [43]. The problem is that feature-oriented languages and tools involve usually a code generation step during composition in that they transform code into a lower-level representation. For example, the AHEAD Tool Suite transforms feature-oriented Jak code into object-oriented Java code by translating refinements of classes into subclasses [10]. Other languages and tools work similarly [7, 5, 9].

A problem of these languages and tools is that errors can be detected only at compilation time, not at composition time. While the compiler may detect errors caused by improper feature composition, it cannot recognize the actual cause of their occurrence. For example, a feature may refer to a class that is not present because the feature the class belongs to is not present in a program variant, or a feature may affect a program element that is being introduced in a subsequent development step, which violates the principle of stepwise refinement. The problem is that information about features and their composition is lost during translation to the lower-level representation. Knowledge about features would help to identify an improper feature composition and to create a precise error message.

Consequently, Thaker et al. suggested the development of a type system for feature-oriented languages and tools that can be used to check for the above errors at composition time. We present such a formal type system along with a soundness proof. To this end, we develop a calculus for a simple feature-oriented language on top of Featherweight Java (FJ) [23], called Feature Featherweight Java (FFJ). The syntax and semantics of FFJ conform to common feature-oriented languages. The type system not only incorporates language constructs for feature composition, but also guarantees that the principle of stepwise refinement is not violated.

FFJ is interesting insofar as it is concerned partly with the programming language level (it provides language constructs for class, method, and constructor refinement on top of FJ) and partly with the composition engine at the meta-level (it relies on information about features that is collected outside the program text during

GPCE'08, October 19–23, 2008, Nashville, Tennessee, USA.

composition). Prior work on mixins, traits, and virtual classes did not make this distinction, which is discussed in Section 5.

2. An Overview of FFJ

Before we go into detail, we give an informal overview of FFJ. FFJ builds on FJ. FJ is a language that models a minimal subset of Java. FJ provides basic constructs like classes, fields, methods, and inheritance, but it does not support interfaces, exceptions, access modifiers, overloading, etc., not even assignment. In fact, FJ represents a functional core of Java and, therefore, every FJ program is also a regular Java program, but not vice versa.

FFJ extends FJ by new language constructs for feature composition and by corresponding evaluation and type rules. Although it is an extension of FJ, FFJ's key innovations can be used with other languages, e.g., with C#.

As with other feature-oriented languages [10, 7, 9], the notion of a feature does not appear in the language syntax. That is, the programmer does not state explicitly in the program text that a class or method belongs to a feature. Features are merely represented by containment hierarchies that are directories which aggregate the code artifacts belonging to a feature [10]. This is necessary since a feature may contain, beside code, also further supporting documents, e.g., documentation, test cases, or design documents. By superimposing containment hierarchies, the code artifacts of different features are merged [10, 9].

In FFJ, a programmer can add new classes to a program via the introduction of a new feature, which is trivial since only a new class file with a distinct name has to be supplied by the feature's containment hierarchy. Furthermore, using a feature, one can extend an existing class by a class refinement. A class refinement is declared like a class but prefixed with the keyword refines. During composition, classes and refinements with the same name are merged, i.e., their members are merged. Finally, a class refinement may refine existing methods, which is similar to overriding a superclass' method by a subclass' method.

The distinction between code artifacts (classes and class refinements) and features (containment hierarchies) requires a special treatment in FFJ's semantics and type system, which is different from prior work (see Section 5). Consequently, we use in the type system information about features that has been collected by the composition engine and that does not appear in the program text.

We begin with a description of basic FFJ, a language that provides constructs for feature composition, and proceed with four extensions for stepwise refinement. The extensions are largely orthogonal and can be combined individually and in any order with basic FFJ.

2.1 Basic FFJ

Figure 1 depicts a simple FFJ program that implements an expression evaluator,[1] which is a solution to the infamous "expression problem".[2] It consists of three features, whose implementations are separated by horizontal lines. The feature *Add* is a basic feature that supports only addition of simple expressions; it introduces two classes Expr and Add (Lines 1–7). The feature *Sub* adds support for subtraction by introducing a class Sub (Lines 8–11). The feature *Eval* adds to each class a method eval for expression evaluation (Lines 12–23).

As usual in FOP, the actual composition specification is not part of the program but expressed using a separate composition specification, called a feature expression [10, 8]. Knowledge about the feature composition order is managed by the composition engine and used for evaluation and typing.

```
1  class Expr extends Object {        // Feature Add ...
2    Expr() { super(); }
3  };
4  class Add extends Expr {
5    Expr a; Expr b;
6    Add(Expr a, Expr b) { super(); this.a=a; this.b=b; }
7  }
8  class Sub extends Expr {           // Feature Sub ...
9    Expr a; Expr b;
10   Sub(Expr a, Expr b) { super(); this.a=a; this.b=b; }
11 }
12 refines class Expr {               // Feature Eval ...
13   refines Expr() { original(); }
14   int eval() { return 0; }
15 }
16 refines class Add {
17   refines Add(Expr a, Expr b) { original(a,b); }
18   refines int eval() { return this.a.eval()+this.b.eval(); }
19 }
20 refines class Sub {
21   refines Sub(Expr a, Expr b) { original(a,b); }
22   refines int eval() { return this.a.eval()−this.b.eval(); }
23 }
```

Figure 1. A solution to the "expression problem" in FFJ.

The above example illustrates the main constructs of FFJ. Like in FJ, an FFJ program consists of a set of classes that, in turn, contain a single constructor each, as well as methods and fields. Unlike in FJ, an FFJ program may contain class refinements each of which refine an existing class, which we call henceforth the *base class* of the refinement. A class refinement contains a constructor refinement, a set of methods and fields that are added to the base class, and a set of method refinements that refine the base class' methods. A base class, along with its refinements, has the semantics of a compound class that contains all fields and methods of its constituents, and constructor refinements extend the base class' constructor and method refinements replace base class' methods. That is, class refinements add new and change or extend existing members. Note that it is not possible to instantiate a base class in isolation, but only together with all refinements applied subsequently.

A feature may contain several classes and class refinements, but we impose some restrictions. First, for the sake of consistency, a feature is not allowed to introduce a class that is already present in a program it is composed with. Second, a feature is not allowed two apply two refinements to the same class because, otherwise, the order of applying class refinements would be arbitrary.

Like in FJ, each class must declare exactly one superclass, which may be Object. In contrast, a class refinement does not declare (additional) superclasses. Later on, we will extend FFJ such that class refinements declare further superclasses. Again, with 'base class' we refer to the class that is refined by a class refinement and with 'superclass' we refer to the class that is extended by a subclass.

Typically, with a sequence of features, a programmer can apply several refinements to a class, which is called a *refinement chain*. A refinement that is applied immediately before another refinement in the chain is called its *predecessor*. Conversely, a class refinement that is applied immediately after another refinement is called its *successor*. The order of refinements in a refinement chain is determined by the selection of features and their composition order. Figure 2 depicts the refinement and inheritance relationships of our expression example.

Fields are unique within the scope of a class and its inheritance hierarchy and refinement chain. That is, a refinement or subclass

[1] Although not part of FJ and FFJ, we use basic data types, constants, and operators in our examples for sake of comprehensibility.

[2] The expression problem was named by Phil Wadler in 1998 but has been known for many years [41, 15]; see Torgerson [44] for a retrospective overview.

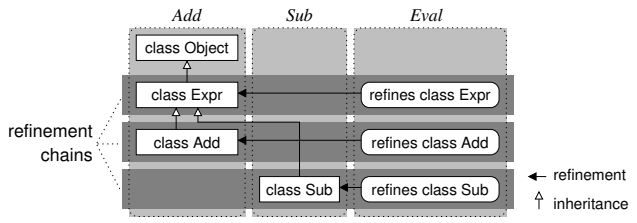

Figure 2. Relationships in the expression example.

is not allowed to add a field that is already defined. With methods this is different. A property that FFJ has inherited from FJ is that subclasses may override methods of superclasses. Similarly to FJ, FFJ does not allow the programmer to use super inside a method body, as is allowed in Java. That is, method overriding in FFJ means essentially method replacement.

A method in FFJ (and FJ) is similar to a Java method except that its body is an expression (prefixed with return) and not a sequence of statements. This is due to the functional nature of FFJ (and FJ). Furthermore, overloading of methods (methods with equal names and different argument types) is not allowed in FFJ (and FJ).

Unlike classes, class refinements are not allowed to define methods that have been defined before in the refinement chain. That is, class refinements cannot override methods. This is to avoid inadvertent replacement. But, instead, a class refinement may declare a method refinement using the keyword refines. This enables the type checker to recognize the difference between method refinement and inadvertent overriding/replacement and, possibly, to warn the programmer. Note that refinements may also refine methods that have been introduced by any superclass of the base class or by previous refinements of the refinement chain. The difference between method refinement and method overriding becomes more useful in an extension of basic FFJ that allows a method refinement to reuse the body of a refined method (see Section 2.2).

As shown in Figure 2, refinement chains grow from left to right and inheritance hierarchies from top to bottom. When looking up a method body, FFJ traverses the combined inheritance and refinement hierarchy of the object the method belongs to and selects the right-most and bottom-most body of a method declaration or method refinement that is compatible. That is, first, FFJ looks for a method declaration or method refinement in the refinement chain of the object's class, starting with the last refinement back to the class declaration itself. The first body of a matching method declaration or method refinement is returned. If the method is not found in the class' refinement chain or in its declaration, the methods in the superclass (and then the superclass' superclass, etc.) are searched, each again from latest refinement to the class declaration itself.

Finally, each class must declare exactly one constructor that is used solely to initialize the class' fields. Similarly, a class refinement must declare exactly one constructor refinement that initializes the class refinement's fields. A constructor expects values for all fields that have been declared by its class and the class' superclasses. The values for the superclass are passed via super. Similarly, a constructor refinement expects values for all fields that have been declared by its class refinement, by previous refinements in the refinement chain, and by the base class. The values for the predecessor in the refinement chain are passed via original.

2.2 Extensions for Stepwise Refinement

Method Extension

In FOP, the replacement of methods is considered inelegant [43]. In FFJ, method refinements may override methods and effectively replace them (same for method overriding in subclasses). This prevents programmers from *extending* existent functionality and leads to code replication in that programmers repeat code of extended methods. In order to foster extension [43], we allow programmers to invoke the refined method from the method refinement, which has been first used in the BETA programming language [32]. The keyword original is used to refer to the refined method from within the refining method, and it may occur multiple times but at least once; otherwise a warning or error is reported.

For example, a feature might refine the eval methods of our expression evaluator in order to log their invocation. Using original, the method refinement of eval can extend the existing method instead of replacing it:

```
1 refines class Add {                    // Feature Logging ...
2   refines int eval() { return new Log().write(original()); }
3 } ...
```

The method write expects an integer, logs its value, and returns the integer unchanged.

Default Values

A class refinement in FFJ may add new fields to a class and the corresponding constructor refinement extends the class' constructor initializing these fields. The problem of this simple mechanism is that a class cannot be used anymore by client classes that have been added before the class refinement in question. This is because the class refinement extends the constructor's signature and, for a client class that has been introduced before, knowing about these new fields in the first place is unlikely.

Suppose a refinement of Expr that adds a new field and that refines the base class' constructor accordingly:

```
1 refines class Expr {
2   int id;
3   refines Expr(int id) { original(); this.id=id; }
4 }
```

Applying this refinement breaks the constructor of Add; the constructor's call of super receives an empty list of arguments, whereas the refined constructor of Expr expects an integer.

There are four options of solving this problem: (1) the constructor of Add can be modified expecting a value for id, (2) the constructor of Add can be refined expecting a value for id that is passed to Expr via super, (3) a class may have multiple constructors, or (4) the field id can be initialized with some sort of default value. Option (1) requires invasive changes to earlier features (violates the principle of stepwise refinement), (2) requires additional effort in refining clients of the class, and (3) would be possible but change the FJ core calculus significantly. Therefore, we choose the fourth option: providing default values for uninitialized fields.

When instantiating a class, a programmer does not need to pass values for all arguments of the class' constructor but only for some of them (i.e., for a subsequence of the argument list). The remaining arguments are filled with default values supplied in another way by the programmer or by the type system. As we will see later on, default values can be supplied by the programmer before invoking the constructor or even generated automatically.

Superclass Declaration

In basic FFJ, class refinements may add new field and method declarations and refine existing methods. A practice that has proved useful in stepwise refinement is that subsequent features may also alter the inheritance hierarchy [39]. Therefore, in an extension of FFJ, we let each class refinement declare a superclass, much like a class declaration. For example, we can refine the class Add in order to inherit additionally from Comparable:

```
1  class Comparable extends Object {
2    Comparable() { super(); }
3    boolean equals(Comparable c) { return true; }
4  }
5  refines class Add extends Comparable {
6    refines Add(Expr a, Expr b) { super(); original(a, b); }
7    refines boolean equals(Comparable c) {
8      return ((Add)c).a == this.a && ((Add)c).b == this.b;
9    }
10 }
```

Note that Add inherits now from both Expr and Comparable. In order to pass arguments properly, the constructor of Add's refinement uses super for passing arguments to the superclass and original for passing arguments to the base class.

Effectively, a class that is merged with its class refinements inherits from multiple classes. However, our intention is not to solve the tricky problems of multiple inheritance [42], so we impose some restrictions. First, a class refinement is only allowed to declare a superclass that has not been declared before in the refinement chain, except for Object. Second, all further superclasses of this superclass must not be declared before. Third, the superclass (incl. all its superclasses) must not introduce a field or method that has been introduced before in the refinement chain. All these conditions avoid name clashes and ambiguities.

Backward References

Finally, we enable the type system to check whether all classes, class refinements, methods, method refinements, and fields contain only references to features that have been added before, which we call *backward references*. In contrast, the type checker rejects programs containing *forward references*. This is in line with the principle of stepwise refinement disallowing code of previous development steps to affect code of subsequent development steps [45, 31].

For example, in Figure 1, the base class Add may contain a reference to the class Expr but must not contain a reference to the class Sub and its members because they are being introduced subsequently. We can enforce this property by checking superclass and field declarations, as well as bodies and signatures of methods and method refinements for the direction of their type or member references.

3. The Basic FFJ Calculus

In this section, we describe the syntax, evaluation, and type rules of basic FFJ. For a better understanding of the changes and extensions that FFJ makes to FJ, in the colored version of the paper, we highlight modified rules with shaded yellow boxes and new rules with shaded purple boxes.

3.1 Syntax

In Figure 3, we depict the syntax of FFJ, which is a straightforward extension of the syntax of FJ [23]. An FFJ program consists of a set of class and refinement declarations, an expression, and information about features collected externally by the composition engine. As mentioned, the actual composition specification is not part of the program but expressed with a separate composition specification.

A class declaration CD contains a list $\overline{\mathsf{C}}\ \overline{\mathsf{f}}$ of fields,[3] a constructor declaration KD, and list $\overline{\mathsf{MD}}$ of method declarations. A class refinement CR contains a list $\overline{\mathsf{C}}\ \overline{\mathsf{f}}$ of fields, a constructor refinement KR, a list $\overline{\mathsf{MD}}$ of method declarations, and a list $\overline{\mathsf{MR}}$ of

[3] We abbreviate lists in the obvious way: $\overline{\mathsf{C}}\ \overline{\mathsf{f}}$ is shorthand for $\mathsf{C}_1\ \mathsf{f}_1, \ldots, \mathsf{C}_n\ \mathsf{f}_n$; $\overline{\mathsf{C}}\ \overline{\mathsf{f}}$; is shorthand for $\mathsf{C}_1\ \mathsf{f}_1; \ldots; \mathsf{C}_n\ \mathsf{f}_n$; and $\mathsf{this}.\overline{\mathsf{f}}=\overline{\mathsf{f}}$; is shorthand for $\mathsf{this}.\mathsf{f}_1=\mathsf{f}_1; \ldots; \mathsf{this}.\mathsf{f}_n=\mathsf{f}_n;$.

Navigating the refinement chain

$$\frac{\mathsf{S}\text{ is the successor of }\mathsf{R}}{succ(\mathsf{R}) = \mathsf{S}} \qquad \frac{\mathsf{S}\text{ is the predecessor of }\mathsf{R}}{pred(\mathsf{R}) = \mathsf{S}}$$

Subtyping

$$\boxed{\mathsf{C} <: \mathsf{D}}$$

$$\mathsf{C} <: \mathsf{C} \qquad \frac{\mathsf{C} <: \mathsf{D} \qquad \mathsf{D} <: \mathsf{E}}{\mathsf{C} <: \mathsf{E}}$$

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ldots\}}{\mathsf{C} <: \mathsf{D}}$$

Figure 4. Subtyping and refinement in basic FFJ.

method refinements; its declaration is prefixed with the keyword refines. Method and constructor declarations are taken from FJ without change: A method m expects arguments $\overline{\mathsf{C}}\ \overline{\mathsf{x}}$, contains a body return t, and returns a result of type C. A constructor expects two lists $\overline{\mathsf{D}}\ \overline{\mathsf{g}}$ and $\overline{\mathsf{C}}\ \overline{\mathsf{f}}$ of arguments for the fields of the superclass (passed via $\mathsf{super}(\overline{\mathsf{g}})$) and for the fields of its own class (initialized via $\mathsf{this}.\overline{\mathsf{f}}=\overline{\mathsf{f}}$). A constructor refinement KR expects arguments for the predecessor refinement (passed via $\mathsf{original}(\overline{\mathsf{h}})$) and for its own fields ($\mathsf{this}.\overline{\mathsf{f}}=\overline{\mathsf{f}}$). A method refinement is much like a method declaration; constructor and method refinements begin with the keyword refines. The remaining syntax rules for terms t and values v are straightforward and taken from FJ without change.

Class names (meta-variables A–E) are simple identifiers. A refinement (meta-variables R–T) is identified by the name of the base class C and the name of the feature F it belongs to. Declarations of classes and refinements can be looked up via the class table CT. As in FJ, we impose some sanity conditions on the class table: (1) $CT(\mathsf{C}) = \mathsf{class\ C}\ldots$ for every $\mathsf{C} \in dom(CT)$; (2) $\mathsf{Object} \notin dom(CT)$; (3) for every class name C (except Object) appearing anywhere in CT, we have $\mathsf{C} \in dom(CT)$; and (4) there are no cycles (incl. self-cycles) in the inheritance relation. The conditions for class refinements are analogous.

3.2 Subtyping and Refinement

In Figure 4, we show the refinement and subtyping relations of FFJ. There are two auxiliary functions that return the next refinement ($succ$) and the previous refinement ($pred$) in a refinement chain. These definitions rely on information collected by the composition engine, e.g., the features' composition order. For simplicity, the two functions may be used with both classes and class refinements, and they return Object if there is no class refinement that matches. Finally, shown on the bottom, there is a subtype relation <: identical to the one of FJ.

3.3 Auxiliary Definitions

For FFJ, we have modified some auxiliary definitions of FJ, and we have added some, as shown in Figure 5. The function *fields* returns the fields of a class including the fields of its subclasses and, in extension to FJ, the fields added by its refinements. The function *rfields* is similar except that the refinement chain is searched from right to left. This is useful later to determine the fields that have been introduced before a given refinement. The function *mtype* returns the type of a method. In contrast to FJ, in FFJ first the refinement chain is searched from left to right and, if an appropriate method is not found, the search is continued in the corresponding superclass. Like *rfields*, the function *rmtype* is used to look for a method type from right to left in a refinement chain. Function *mbody* looks up the most specific and most refined method body. That is, it returns the method body that is right-most and bottom-

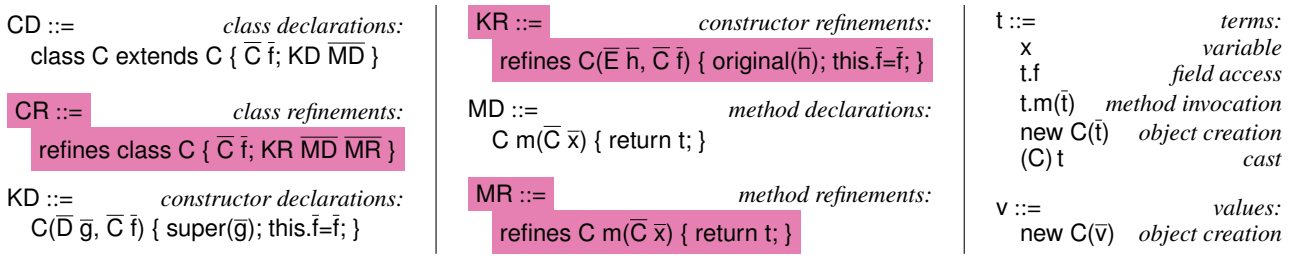

Figure 3. Syntax of basic FFJ.

most in the combined refinement and inheritance hierarchy of an object, as explained in Section 2.1. The function *override* establishes whether a method of a superclass is appropriately overridden in a subclass, i.e., whether their signatures match. The function *introduce* establishes whether a method introduced by a class refinement has not been introduced before in its refinement chain. Finally, the function *extends* establishes whether a method refinement refines a method properly, i.e., whether a corresponding method has been introduced before and their signatures match.

Note, for all auxiliary functions f, applications of $f(x)$ or $\neg f(x)$ mean the function is defined or undefined for the value x.

3.4 Evaluation

The evaluation rules of FFJ, shown in Figure 6, are identical to those in FJ – the differences between FJ and FFJ are entirely implemented by the extended/modified auxiliary functions. In rule E-ProjNew, *fields* looks up all fields of a class, including the fields of its refinements. The fact that each class refinement must refine the constructor makes sure that the number of supplied values in a class instantiation is equal to the number of fields that *fields* returns. With the 'default values' extension, this will be different (see Section 4.2). In rule E-InvkNew, *mbody* returns the appropriate method body also considering method refinements, so nothing changes compared to FJ. The remaining rules are straightforward and equal in FJ and FFJ.

3.5 Typing

Figure 7 displays the type rules of FFJ. Rules for term typing are the same as in FJ (left side). Again, this was possible since we changed the auxiliary functions incorporating class, method, and constructor refinement properly. However, we extended the well-formedness rules (right side). The two rules for well-formed methods enforce that the type of the method body's term is a subtype of the declared return type, that a method of a superclass is being overridden appropriately, and that no subsequent refinement introduces a method with the same name. That is, *override* considers methods that are overridden by the given method and *introduce* considers methods that are introduced later and establishes whether they replace the given method or not. The well-formedness rule of method refinement enforces, beside the standard properties, that a corresponding method is being introduced before (and not in the same class refinement) and that the signatures of the two methods match. The well-formedness rule for classes is similar to that of FJ. It enforces the well-formedness of the constructor, the fields, and the methods. The well-formedness rule for class refinements enforces, in addition, that all method refinements are well-formed and that appropriate values are passed for the fields of the refinement and its predecessor.

Note that FJ's type system is not modular such that features can be checked in isolation. The reason is that the auxiliary functions search all classes and refinements. Modular type checking has been addressed in prior work on module systems [34, 21], and it is part of further work to explore the implications for FOP.

3.6 Type Soundness

FFJ is type sound. We state this with the standard theorems Preservation and Progress [46]. The proofs are the same as the one of FJ [23], except for some minor modifications. This is the case because the changes and extensions basic FFJ makes to FJ are largely concerned with the method and field lookup and do not interfere too much with the evaluation order and type system. In some of our extensions this will be different, as we will explain in the next section. See our technical report for the complete proof and further explanations [6].

4. Extensions for Stepwise Refinement

In this section, we integrate each extension individually into FFJ, obtaining FFJ$_{ME}$ for method extensions, FFJ$_{DV}$ for default values, FFJ$_{SD}$ for superclass declarations, and FFJ$_{BR}$ for backward references. For each extension, we show how the syntax, evaluation, and type rules change and if and how the type soundness proof is affected. As the extensions are largely orthogonal, they can be combined freely to obtain a consistent and type sound variant of FFJ.

4.1 FFJ$_{ME}$—Method Extension

In basic FFJ, a method refinement replaces the body of the method that is refined. In FFJ$_{ME}$, method bodies must invoke original (at least once); otherwise a method is not well-formed. The keyword original refers to the method that is refined.

First, we extend the syntax such that original may occur in terms:

$$\begin{array}{lll} \mathsf{t} ::= & \ldots & \textit{basic FFJ terms} \\ & \mathsf{original}(\bar{\mathsf{t}}) & \textit{original invocation} \end{array}$$

Second, we modify *mbody* such that it substitutes every occurrence of original with the method body that is being refined (arguments are renamed); if the refined body contains original in turn, the process is repeated. The evaluation rule E-ProjInvk (Figure 6) is divided into two new rules as follows:

$$\frac{\mathit{mbody}(\mathsf{m},\mathsf{C}) = (\bar{\mathsf{x}},\mathsf{t}_0) \qquad \mathit{succ}(\mathsf{C}) = \mathsf{Object}}{\begin{array}{c}(\mathsf{new\ C}(\bar{\mathsf{v}})).\mathsf{m}(\bar{\mathsf{u}}) \longrightarrow \\ [\bar{\mathsf{x}} \mapsto \bar{\mathsf{u}}, \mathsf{this} \mapsto \mathsf{new\ C}(\bar{\mathsf{v}})]\,\mathsf{t}_0\end{array}} \quad \text{(E-ProjInvk1)}$$

$$\frac{\begin{array}{c}\mathit{mbody}(\mathsf{m},\mathsf{C}) = (\bar{\mathsf{x}},\mathsf{t}_0) \\ \mathsf{R} \text{ is the final refinement of } \mathsf{C}\end{array}}{\begin{array}{c}(\mathsf{new\ C}(\bar{\mathsf{v}})).\mathsf{m}(\bar{\mathsf{u}}) \longrightarrow \\ [\bar{\mathsf{x}} \mapsto \bar{\mathsf{u}}, \mathsf{this} \mapsto \mathsf{new\ C}(\bar{\mathsf{v}})]\,\mathit{eval}(\mathsf{m},\mathsf{R},\mathsf{t}_0)\end{array}} \quad \text{(E-ProjInvk2)}$$

The latter rule uses an auxiliary function *eval* that performs the actual substitution. It begins with the final refinement R in the refinement chain of C and searches the method bodies for original. Each occurrence of original is substituted with the method body

Field lookup $\boxed{\mathit{fields}(\mathsf{C}) = \overline{\mathsf{C}}\ \overline{\mathsf{f}}}$

$$\mathit{fields}(\mathsf{Object}) = \bullet$$

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}}}{\mathit{fields}(\mathsf{C}) = \mathit{fields}(\mathsf{D}),\ \overline{\mathsf{C}}\ \overline{\mathsf{f}},\ \mathit{fields}(\mathit{succ}(\mathsf{C}))}$$

$$\frac{CT(\mathsf{R}) = \mathsf{refines\ class\ C\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}}}{\mathit{fields}(\mathsf{R}) = \overline{\mathsf{C}}\ \overline{\mathsf{f}},\ \mathit{fields}(\mathit{succ}(\mathsf{R}))}$$

Reverse field lookup $\boxed{\mathit{rfields}(\mathsf{R}) = \overline{\mathsf{C}}\ \overline{\mathsf{f}}}$

$$\mathit{rfields}(\mathsf{Object}) = \bullet$$

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}}}{\mathit{rfields}(\mathsf{C}) = \mathit{fields}(\mathsf{D}),\ \overline{\mathsf{C}}\ \overline{\mathsf{f}}}$$

$$\frac{CT(\mathsf{R}) = \mathsf{refines\ class\ C\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}}}{\mathit{rfields}(\mathsf{R}) = \mathit{rfields}(\mathit{pred}(\mathsf{R})),\ \overline{\mathsf{C}}\ \overline{\mathsf{f}}}$$

Method type lookup $\boxed{\mathit{mtype}(\mathsf{m}, \mathsf{C}) = \overline{\mathsf{C}} \to \mathsf{C}}$

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}}\quad \mathsf{B\ m}(\overline{\mathsf{B}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return\ t;}\ \} \in \overline{\mathsf{MD}}}{\mathit{mtype}(\mathsf{m}, \mathsf{C}) = \overline{\mathsf{B}} \to \mathsf{B}}$$

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}}\quad \mathsf{m\ is\ not\ defined\ in}\ \overline{\mathsf{MD}}\quad \mathit{mtype}(\mathsf{m}, \mathit{succ}(\mathsf{C}))}{\mathit{mtype}(\mathsf{m}, \mathsf{C}) = \mathit{mtype}(\mathsf{m}, \mathit{succ}(\mathsf{C}))}$$

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}}\quad \mathsf{m\ is\ not\ defined\ in}\ \overline{\mathsf{MD}}\quad \neg\mathit{mtype}(\mathsf{m}, \mathit{succ}(\mathsf{C}))}{\mathit{mtype}(\mathsf{m}, \mathsf{C}) = \mathit{mtype}(\mathsf{m}, \mathsf{D})}$$

$$\frac{CT(\mathsf{R}) = \mathsf{refines\ class\ C\{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}}\quad \mathsf{B\ m}(\overline{\mathsf{B}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return\ t;}\ \} \in \overline{\mathsf{MD}}}{\mathit{mtype}(\mathsf{m}, \mathsf{R}) = \overline{\mathsf{B}} \to \mathsf{B}}$$

$$\frac{CT(\mathsf{R}) = \mathsf{refines\ class\ C\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}}\quad \mathsf{m\ is\ not\ defined\ in}\ \overline{\mathsf{MD}}}{\mathit{mtype}(\mathsf{m}, \mathsf{R}) = \mathit{mtype}(\mathsf{m}, \mathit{succ}(\mathsf{R}))}$$

Reverse method type lookup $\boxed{\mathit{rmtype}(\mathsf{m}, \mathsf{C}) = \overline{\mathsf{C}} \to \mathsf{C}}$

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}}\quad \mathsf{B\ m}(\overline{\mathsf{B}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return\ t;}\ \} \in \overline{\mathsf{MD}}}{\mathit{rmtype}(\mathsf{m}, \mathsf{C}) = \overline{\mathsf{B}} \to \mathsf{B}}$$

$$\frac{CT(\mathsf{R}) = \mathsf{refines\ class\ C\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}}\quad \mathsf{B\ m}(\overline{\mathsf{B}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return\ t;}\ \} \in \overline{\mathsf{MD}}}{\mathit{rmtype}(\mathsf{m}, \mathsf{R}) = \overline{\mathsf{B}} \to \mathsf{B}}$$

$$\frac{CT(\mathsf{R}) = \mathsf{refines\ class\ C\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}}\quad \mathsf{m\ is\ not\ defined\ in}\ \overline{\mathsf{MD}}}{\mathit{rmtype}(\mathsf{m}, \mathsf{R}) = \mathit{rmtype}(\mathsf{m}, \mathit{pred}(\mathsf{R}))}$$

Method body lookup $\boxed{\mathit{mbody}(\mathsf{m}, \mathsf{C}) = (\overline{\mathsf{x}}, \mathsf{t})}$

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}}\quad \mathsf{B\ m}(\overline{\mathsf{B}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return\ t;}\ \} \in \overline{\mathsf{MD}}\quad \neg\mathit{mbody}(\mathsf{m}, \mathit{succ}(\mathsf{C}))}{\mathit{mbody}(\mathsf{m}, \mathsf{C}) = (\overline{\mathsf{x}}, \mathsf{t})}$$

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}}\quad \mathsf{m\ is\ not\ defined\ in}\ \overline{\mathsf{MD}}\quad \neg\mathit{mbody}(\mathsf{m}, \mathit{succ}(\mathsf{C}))}{\mathit{mbody}(\mathsf{m}, \mathsf{C}) = \mathit{mbody}(\mathsf{m}, \mathsf{D})}$$

$$\mathit{mbody}(\mathsf{m}, \mathsf{C}) = \mathit{mbody}(\mathsf{m}, \mathit{succ}(\mathsf{C}))$$

$$\frac{CT(\mathsf{R}) = \mathsf{refines\ class\ C\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}}\quad \mathsf{B\ m}(\overline{\mathsf{B}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return\ t;}\ \} \in \overline{\mathsf{MD}}\ \text{ or }\ \mathsf{refines\ B\ m}(\overline{\mathsf{B}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return\ t;}\ \} \in \overline{\mathsf{MR}}\quad \neg\mathit{mbody}(\mathsf{m}, \mathit{succ}(\mathsf{R}))}{\mathit{mbody}(\mathsf{m}, \mathsf{R}) = (\overline{\mathsf{x}}, \mathsf{t})}$$

$$\frac{CT(\mathsf{R}) = \mathsf{refines\ class\ C\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}}}{\mathit{mbody}(\mathsf{m}, \mathsf{R}) = \mathit{mbody}(\mathsf{m}, \mathit{succ}(\mathsf{R}))}$$

Valid method overriding $\boxed{\mathit{override}(\mathsf{m}, \mathsf{D}, \overline{\mathsf{C}} \to \mathsf{C}_0)}$

$$\frac{\mathit{mtype}(\mathsf{m}, \mathsf{D}) = \overline{\mathsf{D}} \to \mathsf{D}_0\ \text{ implies }\ \overline{\mathsf{C}} = \overline{\mathsf{D}}\ \text{ and }\ \mathsf{C}_0 = \mathsf{D}_0}{\mathit{override}(\mathsf{m}, \mathsf{D}, \overline{\mathsf{C}} \to \mathsf{C}_0)}$$

Valid method introduction $\boxed{\mathit{introduce}(\mathsf{m}, \mathsf{C})}$

$$\frac{\neg\mathit{mtype}(\mathsf{m}, \mathit{succ}(\mathsf{C}))}{\mathit{introduce}(\mathsf{m}, \mathsf{C})}$$

Valid method refinement $\boxed{\mathit{extend}(\mathsf{m}, \mathsf{R}, \overline{\mathsf{C}} \to \mathsf{C}_0)}$

$$\frac{\mathit{rmtype}(\mathsf{m}, \mathit{pred}(\mathsf{R})) = \overline{\mathsf{B}} \to \mathsf{B}_0\ \text{ implies }\ \overline{\mathsf{C}} = \overline{\mathsf{B}}\ \text{ and }\ \mathsf{C}_0 = \mathsf{B}_0}{\mathit{extend}(\mathsf{m}, \mathsf{R}, \overline{\mathsf{C}} \to \mathsf{C}_0)}$$

Figure 5. Auxiliary definitions of basic FFJ.

$$\frac{\mathit{fields}(\mathsf{C}) = \overline{\mathsf{C}}\ \overline{\mathsf{f}}}{(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{f}_i \longrightarrow \mathsf{v}_i} \quad \text{(E-ProjNew)}$$

$$\frac{\mathit{mbody}(\mathsf{m}, \mathsf{C}) = (\overline{\mathsf{x}}, \mathsf{t}_0)}{(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{m}(\overline{\mathsf{u}}) \longrightarrow [\overline{\mathsf{x}} \mapsto \overline{\mathsf{u}}, \mathsf{this} \mapsto \mathsf{new\ C}(\overline{\mathsf{v}})]\,\mathsf{t}_0} \quad \text{(E-ProjInvk)}$$

$$\frac{\mathsf{C} <: \mathsf{D}}{(\mathsf{D})(\mathsf{new\ C}(\overline{\mathsf{v}})) \longrightarrow \mathsf{new\ C}(\overline{\mathsf{v}})} \quad \text{(E-CastNew)}$$

$$\frac{\mathsf{t}_0 \longrightarrow \mathsf{t}_0'}{\mathsf{t}_0.\mathsf{f} \longrightarrow \mathsf{t}_0'.\mathsf{f}} \quad \text{(E-Field)}$$

$$\frac{\mathsf{t}_0 \longrightarrow \mathsf{t}_0'}{\mathsf{t}_0.\mathsf{m}(\overline{\mathsf{t}}) \longrightarrow \mathsf{t}_0'.\mathsf{m}(\overline{\mathsf{t}})} \quad \text{(E-InvkRecv)}$$

$$\frac{\mathsf{t}_i \longrightarrow \mathsf{t}_i'}{\mathsf{v}_0.\mathsf{m}(\overline{\mathsf{v}}, \mathsf{t}_i, \overline{\mathsf{t}}) \longrightarrow \mathsf{v}_0.\mathsf{m}(\overline{\mathsf{v}}, \mathsf{t}_i', \overline{\mathsf{t}})} \quad \text{(E-InvkArg)}$$

$$\frac{\mathsf{t}_i \longrightarrow \mathsf{t}_i'}{\mathsf{new\ C}(\overline{\mathsf{v}}, \mathsf{t}_i, \overline{\mathsf{t}}) \longrightarrow \mathsf{new\ C}(\overline{\mathsf{v}}, \mathsf{t}_i', \overline{\mathsf{t}})} \quad \text{(E-NewArg)}$$

$$\frac{\mathsf{t}_0 \longrightarrow \mathsf{t}_0'}{(\mathsf{C})\mathsf{t}_0.\mathsf{f} \longrightarrow (\mathsf{C})\mathsf{t}_0'.\mathsf{f}} \quad \text{(E-Cast)}$$

Figure 6. Evaluation of basic FFJ.

Term typing $\boxed{\Gamma \vdash \mathsf{t} : \mathsf{C}}$

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

$$\frac{\Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \qquad \mathit{fields}(\mathsf{C}_0) = \overline{\mathsf{C}}\ \overline{\mathsf{f}}}{\Gamma \vdash \mathsf{t}_0.\mathsf{f}_i : \mathsf{C}_i} \quad \text{(T-Field)}$$

$$\frac{\Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \qquad \mathit{mtype}(\mathsf{m}, \mathsf{C}_0) = \overline{\mathsf{D}} \to \mathsf{C} \qquad \Gamma \vdash \overline{\mathsf{t}} : \overline{\mathsf{C}} \qquad \overline{\mathsf{C}} <: \overline{\mathsf{D}}}{\Gamma \vdash \mathsf{t}_0.\mathsf{m}(\overline{\mathsf{t}}) : \mathsf{C}} \quad \text{(T-Invk)}$$

$$\frac{\mathit{fields}(\mathsf{C}) = \overline{\mathsf{D}}\ \overline{\mathsf{f}} \qquad \Gamma \vdash \overline{\mathsf{t}} : \overline{\mathsf{C}} \qquad \overline{\mathsf{C}} <: \overline{\mathsf{D}}}{\Gamma \vdash \mathsf{new\ C}(\overline{\mathsf{t}}) : \mathsf{C}} \quad \text{(T-New)}$$

$$\frac{\Gamma \vdash \mathsf{t}_0 : \mathsf{D} \qquad \mathsf{D} <: \mathsf{C}}{\Gamma \vdash (\mathsf{C})\mathsf{t}_0 : \mathsf{C}} \quad \text{(T-UCast)}$$

$$\frac{\Gamma \vdash \mathsf{t}_0 : \mathsf{D} \qquad \mathsf{C} <: \mathsf{D} \qquad \mathsf{C} \neq \mathsf{D}}{\Gamma \vdash (\mathsf{C})\mathsf{t}_0 : \mathsf{C}} \quad \text{(T-DCast)}$$

$$\frac{\Gamma \vdash \mathsf{t}_0 : \mathsf{D} \qquad \mathsf{C} \not<: \mathsf{D} \qquad \mathsf{D} \not<: \mathsf{C} \qquad \textit{stupid warning}}{\Gamma \vdash (\mathsf{C})\mathsf{t}_0 : \mathsf{C}} \quad \text{(T-SCast)}$$

Method typing $\boxed{\mathsf{MD\ OK\ in\ C/R}}$

$$\frac{\overline{\mathsf{x}} : \overline{\mathsf{C}}, \mathsf{this} : \mathsf{C} \vdash \mathsf{t}_0 : \mathsf{E}_0 \qquad \mathsf{E}_0 <: \mathsf{C}_0 \qquad CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ldots\} \qquad \mathit{override}(\mathsf{m}, \mathsf{D}, \overline{\mathsf{C}} \to \mathsf{C}_0) \qquad \mathit{introduce}(\mathsf{m}, \mathsf{C})}{\mathsf{C}_0\ \mathsf{m}(\overline{\mathsf{C}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return\ t}_0;\ \}\ \mathsf{OK\ in\ C}}$$

$$\frac{\overline{\mathsf{x}} : \overline{\mathsf{C}}, \mathsf{this} : \mathsf{C} \vdash \mathsf{t}_0 : \mathsf{E}_0 \qquad \mathsf{E}_0 <: \mathsf{C}_0 \qquad CT(\mathsf{R}) = \mathsf{refines\ class\ C}\ \{\ldots\} \qquad \mathit{introduce}(\mathsf{m}, \mathsf{R})}{\mathsf{C}_0\ \mathsf{m}(\overline{\mathsf{C}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return\ t}_0;\ \}\ \mathsf{OK\ in\ R}}$$

Method refinement typing $\boxed{\mathsf{MR\ OK\ in\ R}}$

$$\frac{\overline{\mathsf{x}} : \overline{\mathsf{C}}, \mathsf{this} : \mathsf{C} \vdash \mathsf{t}_0 : \mathsf{E}_0 \qquad \mathsf{E}_0 <: \mathsf{C}_0 \qquad CT(\mathsf{R}) = \mathsf{refines\ class\ C}\ \{\ldots\ \overline{\mathsf{MD}}\ \ldots\} \qquad \mathsf{m\ not\ defined\ in}\ \overline{\mathsf{MD}} \qquad \mathit{extend}(\mathsf{m}, \mathsf{R}, \overline{\mathsf{C}} \to \mathsf{C}_0)}{\mathsf{refines\ C}_0\ \mathsf{m}(\overline{\mathsf{C}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return\ t}_0;\ \}\ \mathsf{OK\ in\ R}}$$

Class typing $\boxed{\mathsf{C\ OK}}$

$$\frac{\mathsf{KD} = \mathsf{C}(\overline{\mathsf{D}}\ \overline{\mathsf{g}}, \overline{\mathsf{C}}\ \overline{\mathsf{f}})\ \{\ \mathsf{super}(\overline{\mathsf{g}});\ \mathsf{this}.\overline{\mathsf{f}}=\overline{\mathsf{f}};\ \} \qquad \mathit{fields}(\mathsf{D}) = \overline{\mathsf{D}}\ \overline{\mathsf{g}} \qquad \overline{\mathsf{MD}}\ \mathsf{OK\ in\ C}}{\mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KD}\ \overline{\mathsf{MD}}\ \}\ \mathsf{OK}}$$

Class refinement typing $\boxed{\mathsf{R\ OK}}$

$$\frac{\mathsf{KR} = \mathsf{C}(\overline{\mathsf{E}}\ \overline{\mathsf{h}}, \overline{\mathsf{C}}\ \overline{\mathsf{f}})\ \{\ \mathsf{original}(\overline{\mathsf{h}});\ \mathsf{this}.\overline{\mathsf{f}}=\overline{\mathsf{f}};\ \} \qquad \mathit{rfields}(\mathit{pred}(\mathsf{R})) = \overline{\mathsf{E}}\ \overline{\mathsf{h}} \qquad \overline{\mathsf{MD}}\ \mathsf{OK\ in\ R} \qquad \overline{\mathsf{MR}}\ \mathsf{OK\ in\ R} \qquad CT(\mathsf{R}) = \mathsf{refines\ class\ C}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KR}\ \overline{\mathsf{MD}}\ \overline{\mathsf{MR}}\ \}}{\mathsf{refines\ class\ C}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KR}\ \overline{\mathsf{MD}}\ \overline{\mathsf{MR}}\ \}\ \mathsf{OK}}$$

Figure 7. Typing in basic FFJ.

that is refined and variables are renamed properly; this recurses until the method body in question does not contain original. Due to the lack of space, the definition of *eval* is relegated to the technical report [6].

Third, we have to add a premise to the well-formedness rule MR OK in R to let the type system make sure that every body of a well-formed method refinement contains a reference to original:

$$\frac{\overline{\mathsf{x}} : \overline{\mathsf{C}}, \mathsf{this} : \mathsf{C} \vdash \mathsf{t}_0 : \mathsf{E}_0 \qquad \mathsf{E}_0 <: \mathsf{C}_0 \qquad CT(\mathsf{R}) = \mathsf{refines\ class}\ \{\ldots\} \qquad \mathit{extend}(\mathsf{m}, \mathsf{R}, \overline{\mathsf{C}} \to \mathsf{C}_0) \qquad \mathsf{t}_0\ \text{contains original}}{\mathsf{refines}\ \mathsf{C}_0\ \mathsf{m}(\overline{\mathsf{C}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return}\ \mathsf{t}_0;\ \}\ \mathsf{OK\ in\ R}}$$

Analogously, we have to add a premise to the two well-formedness rules of method typing in order to reject method declarations whose bodies contain original (see the technical report [6]).

The introduction of original to method bodies does not interfere with the evaluation order and the type system. Function $eval$ substitutes all occurrences of original with the method bodies that are refined, which effectively allows a refinement to extend a method body. As a consequence, method bodies in FJ$_{ME}$ are indistinguishable from the ones in basic FFJ. Evaluation and typing in FJ$_{ME}$ can proceed similarly to basic FFJ. Hence, FJ$_{ME}$ is type sound (see the technical report [6] for more details).

4.2 FJ$_{DV}$—Default Values

In order to include default values, we make some changes to FFJ, obtaining FJ$_{DV}$. First, we allow instantiations of classes to supply only a subsequence of arguments to the constructor. Since the type checker cannot always recognize which formal arguments are meant, such a subsequence must match a prefix of the sequence of expected arguments.

With this mechanism, classes can be instantiated without knowledge of refinements that add new fields subsequently. In order to assign proper values to the fields that have not been initialized, we use default values generated by the FJ$_{DV}$ calculus. Of course, it would also be possible to let the programmer provide the default values, which is done in a related FOP calculus [3], but we generate default values in order to keep the calculus simple.

Generated default values are in some sense the neutral elements of a given type. Only bottom-level classes of the class hierarchy can be instantiated using default values. The reason is that, otherwise, mapping arguments to fields would be ambiguous.

The auxiliary function *default*(C) computes the default value of class C. It does not rely on extra information supplied by the programmer:

$$\mathit{default}(\mathsf{Object}) = \mathsf{new\ Object()}$$

$$\frac{\mathit{fields}(\mathsf{C}) = \overline{\mathsf{C}}\ \overline{\mathsf{f}}}{\mathit{default}(\mathsf{C}) = \mathsf{new\ C}(\mathit{default}(\mathsf{C}_1), \ldots, \mathit{default}(\mathsf{C}_n))}$$

Using default values, we divide the evaluation rule E-ProjNew for projection of Figure 6 into E-ProjNew1 and E-ProjNew2:

$$\frac{\mathit{fields}(\mathsf{C}) = \overline{\mathsf{C}}\ \overline{\mathsf{f}} \quad |\overline{\mathsf{v}}| \geq i}{(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{f}_i \longrightarrow \mathsf{v}_i} \qquad \text{(E-ProjNew1)}$$

$$\frac{\mathit{fields}(\mathsf{C}) = \overline{\mathsf{C}}\ \overline{\mathsf{f}} \quad |\overline{\mathsf{v}}| < i}{(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{f}_i \longrightarrow \mathit{default}(\mathsf{C}_i)} \qquad \text{(E-ProjNew2)}$$

If the sequence of supplied values contains the value of the projected field f_i (E-ProjNew1), nothing changes compared to basic FFJ. On the other hand, if the sequence of supplied values does not contain the value of the projected field f_i (E-ProjNew2), a default value v is supplied.

Finally, we have to update the type rule T-New to allow a smaller number of values to be supplied than the number of fields a class actually contains (incl. its refinements and superclasses):

$$\frac{\mathit{fields}(\mathsf{C}) = \overline{\mathsf{D}}\ \overline{\mathsf{f}}, \overline{\mathsf{E}}\ \overline{\mathsf{h}} \qquad \Gamma \vdash \overline{\mathsf{t}} : \overline{\mathsf{C}} \qquad \overline{\mathsf{C}} <: \overline{\mathsf{D}}}{\Gamma \vdash \mathsf{new\ C}(\overline{\mathsf{t}}) : \mathsf{C}} \qquad \text{(T-New)}$$

The modified evaluation and type rules induce some changes in basic FFJ's type soundness proof in order to carry over to FJ$_{DV}$. Essentially, the cases of instantiations of classes and of projections of fields change such that also subsequences of arguments for a constructor are accepted. In the technical report [6], we explain how the proof changes and show that FJ$_{DV}$ is type sound.

4.3 FJ$_{SD}$—Superclass Declaration

In order to obtain FJ$_{SD}$, we first modify the syntax rules of FFJ such that a class refinement declares a superclass, possibly Object, and a constructor refinement passes the values intended for its superclass via super:

CR ::= *class refinements:*
  refines class C extends D { $\overline{\mathsf{C}}\ \overline{\mathsf{f}}$; KR $\overline{\mathsf{MD}}$ $\overline{\mathsf{MR}}$ }

KR ::= *constructor refinements:*
  refines C($\overline{\mathsf{D}}\ \overline{\mathsf{g}}$, $\overline{\mathsf{E}}\ \overline{\mathsf{h}}$, $\overline{\mathsf{C}}\ \overline{\mathsf{f}}$) { super(g); original($\overline{\mathsf{h}}$); this.$\overline{\mathsf{f}}$=$\overline{\mathsf{f}}$; }

Second, we extend the subtype relation in order to consider also the superclasses of a class that have been declared by its refinements:

$$\frac{CT(\mathsf{R}) = \mathsf{refines\ class\ C\ extends\ D}\ \{\ldots\}}{\mathsf{C} <: \mathsf{D}}$$

Third, we have to modify and extend some auxiliary functions. Now, the function *fields* also collects the fields of the superclasses declared by the class refinements:

$$\frac{CT(\mathsf{R}) = \mathsf{refines\ class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KR}\ \overline{\mathsf{MD}}\ \overline{\mathsf{MR}}\ \}}{\mathit{fields}(\mathsf{C}) = \mathit{fields}(\mathsf{D}),\ \overline{\mathsf{C}}\ \overline{\mathsf{f}},\ \mathit{fields}(\mathit{succ}(\mathsf{C}))}$$

Two new rules for method type and body lookup incorporate also the superclasses of class refinements:

$$\frac{CT(\mathsf{R}) = \mathsf{refines\ class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KR}\ \overline{\mathsf{MD}}\ \overline{\mathsf{MR}}\ \} \qquad \mathsf{m}\ \text{is not defined in}\ \overline{\mathsf{MD}} \qquad \neg \mathit{mtype}(\mathsf{m}, \mathit{succ}(\mathsf{R}))}{\mathit{mtype}(\mathsf{m}, \mathsf{R}) = \mathit{mtype}(\mathsf{m}, \mathsf{D})}$$

$$\frac{CT(\mathsf{R}) = \mathsf{refines\ class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KR}\ \overline{\mathsf{MD}}\ \overline{\mathsf{MR}}\ \} \qquad \mathsf{m}\ \text{is not defined in}\ \overline{\mathsf{MD}}\ \text{or}\ \overline{\mathsf{MR}} \qquad \neg \mathit{mbody}(\mathsf{m}, \mathit{succ}(\mathsf{R}))}{\mathit{mbody}(\mathsf{m}, \mathsf{R}) = \mathit{mbody}(\mathsf{m}, \mathsf{D})}$$

The premises $\neg mtype(\ldots)$ and $\neg mbody(\ldots)$ are necessary to make sure that superclasses are only looked up in the case that there are no matching methods in subsequent class refinements. The remaining rules of the auxiliary functions are simply updated to be compatible with the new syntax of class refinements.

Fourth, the well-formedness rules for methods and class refinements change. Method declarations in class refinements must override methods of the class refinement's superclasses properly:

$$\frac{\overline{\mathsf{x}} : \overline{\mathsf{C}}, \mathsf{this} : \mathsf{C} \vdash \mathsf{t}_0 : \mathsf{E}_0 \qquad \mathsf{E}_0 <: \mathsf{C}_0 \qquad CT(\mathsf{R}) = \mathsf{refines\ class\ C\ extends\ D}\ \{\ldots\} \qquad \mathit{override}(\mathsf{m}, \mathsf{D}, \overline{\mathsf{C}} \to \mathsf{C}_0) \qquad \mathit{introduce}(\mathsf{m}, \mathsf{R})}{\mathsf{C}_0\ \mathsf{m}(\overline{\mathsf{C}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return}\ \mathsf{t}_0;\ \}\ \mathsf{OK\ in\ R}}$$

The well-formedness rule of class refinements enforces additionally that the arguments for the superclass' constructor are

passed properly by a constructor refinement:

$$\frac{\begin{array}{c}\mathsf{KR} \;=\; \mathsf{C}(\overline{\mathsf{D}}\ \overline{\mathsf{g}},\ \overline{\mathsf{E}}\ \overline{\mathsf{h}},\ \overline{\mathsf{C}}\ \overline{\mathsf{f}})\ \{\ \mathsf{super}(\overline{\mathsf{g}});\ \mathsf{original}(\overline{\mathsf{h}});\ \mathsf{this}.\overline{\mathsf{f}}{=}\overline{\mathsf{f}};\ \}\\ \mathit{fields}(\mathsf{D}) \;=\; \overline{\mathsf{D}}\ \overline{\mathsf{g}} \qquad \mathit{rfields}(\mathit{pred}(\mathsf{R})) \;=\; \overline{\mathsf{E}}\ \overline{\mathsf{h}}\\ \overline{\mathsf{MD}}\ \mathsf{OK\ in\ R} \qquad \overline{\mathsf{MR}}\ \mathsf{OK\ in\ R} \qquad \mathit{inherit}(\mathsf{C},\mathsf{R})\\ CT(\mathsf{R}) \;=\; \mathsf{refines\ class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KR}\ \overline{\mathsf{MD}}\ \overline{\mathsf{MR}}\ \}\end{array}}{\mathsf{refines\ class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KR}\ \overline{\mathsf{MD}}\ \overline{\mathsf{MR}}\ \}\ \mathsf{OK}}$$

Note that, using the auxiliary function *inherit*, the rule checks whether a class refinement declares a superclass that has been declared before in the refinement chain and, if this is the case, the corresponding class refinement is not well-formed; the definition of *inherit* can be found in the technical report [6]. The remaining type and evaluation rules are simply updated considering the new syntax of class refinements.

Finally, the type soundness proof changes minimally in that, in the cases of casts, also multiple superclasses are considered. In the technical report [6], we explain how the proof changes and show that FFJ$_{SD}$ is type sound.

4.4 FFJ$_{BR}$—Backward References

In order to disallow forward references in FFJ, we have to modify the well-formedness rules for methods, method refinements, classes, and class refinements, obtaining FFJ$_{BR}$. To this end, we introduce a function $backward$. It establishes either whether a reference to a class/refinement is a backward reference or whether a term contains backward references. The former case is simple, as the compiler simply checks whether a class has been introduced before another class or refinement (i.e., by a previous feature). In the latter case, the given term is traversed and each subterm is checked for backward references to classes (in casts and class instantiations) and to members (in field accesses and method invocations). See the technical report [6] for the definition of $backward$.

We use $backward$ in the well-formedness rules of FFJ$_{BR}$ for methods, method refinements, classes, and class refinements. For brevity, we give here only the rule for method declarations:

$$\frac{\begin{array}{c}\overline{\mathsf{x}} : \overline{\mathsf{C}}, \mathsf{this} : \mathsf{C} \;\vdash\; \mathsf{t}_0 : \mathsf{E}_0 \qquad \mathsf{E}_0 <: \mathsf{C}_0\\ CT(\mathsf{C}) \;=\; \mathsf{class\ C\ extends\ D}\ \{\ldots\}\\ \mathit{override}(\mathsf{m}, \mathsf{D}, \overline{\mathsf{C}} \to \mathsf{C}_0) \qquad \mathit{introduce}(\mathsf{m}, \mathsf{C})\\ \mathit{backward}(\overline{\mathsf{C}}, \mathsf{C}) \qquad \mathit{backward}(\mathsf{C}_0, \mathsf{C}) \qquad \mathit{backward}(\mathsf{t}_0, \mathsf{C})\end{array}}{\mathsf{C}_0\ \mathsf{m}(\overline{\mathsf{C}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return}\ \mathsf{t}_0;\ \}\ \mathsf{OK\ in\ C}}$$

The remaining well-formedness rules are updated analogously.

The modified well-formedness rules of FFJ$_{BR}$ do not interfere with the type soundness proof of basic FFJ [6]. This is easy to see since the well-formedness rules of FFJ$_{BR}$ reject some programs that are well-formed in FFJ. That is, the set of well-formed FFJ$_{BR}$ programs is a subset of the set of well-formed FFJ programs. Consequently, the type soundness theorem also holds for FFJ$_{BR}$, i.e., FFJ$_{BR}$ is type sound.

4.5 Type Soundness of FFJ with all Extensions

We have shown that FFJ$_{ME}$, FFJ$_{DV}$, FFJ$_{SD}$ and FFJ$_{BR}$ are type sound. As the extensions are largely orthogonal, it is easy to show that FFJ with all extensions together is type sound as well. For the complete syntax, evaluation, and typing rules and a type soundness proof we refer the reader to our technical report [6].

4.6 Supported Language Mechanisms

FFJ and its extensions model several mechanisms of feature-oriented languages and tools. In Table 1, we compare FFJ with a selection of these languages, namely Java, FSTComposer [9] and two versions of Jak: an earlier version included in the AHEAD Tool Suite [10], which we call Jak$_1$, and an extended version [43], which we call Jak$_2$.

	FJ	FFJ	FFJ$_{ME}$	FFJ$_{DV}$	FFJ$_{SD}$	FFJ$_{BR}$
Java	√					
Jak$_1$	√	√	√[a]	√[b]		
Jak$_2$	√	√	√	√[b]		√[d]
FSTComposer	√	√	√	√[b]	√[c]	

[a] While Jak$_1$ supports method extensions, in some versions, it does not inform the programmer that a refinement replaces a method.
[b] Jak and FSTComposer support user-defined default values (not generated values).
[c] In FSTComposer, a class refinement may declare new interfaces but not superclasses.
[d] Jak$_2$ does not cover all checks of FFJ$_{BR}$ for disallowing forward references, e.g., method signatures are not checked.

Table 1. Overview of which mechanisms are supported by which calculus and which language or tool.

It is important to note that the purpose of FFJ and its extensions is to reason about properties of feature-oriented languages and tools, like type soundness, formally. The formalizations model only a small subset of their real-world counterparts, though an important one with respect to the type system. In some cases, FFJ is more restrictive than contemporary feature-oriented languages, e.g., FFJ disallows forward references to types referred to from method signatures, which is not enforced by Jak [43].

5. Related Work

Our work on FFJ was motivated by the work of Thaker et al. [43]. They suggested the development of a type system for feature-oriented programming languages and sketched some basic type rules. Furthermore, in their case studies, they found hidden errors using these rules. Nevertheless, their type system is just a sketch, described only informally, and they do not provide a soundness proof. We propose here such a type system, along with a soundness proof, that provides a superset of the rules of Thaker et al.

FFJ is inspired by several feature-oriented languages and tools, most notably AHEAD/Jak [10], FeatureC++ [7], FSTComposer [9], and Prehofer's feature-oriented Java extension [40]. A key aim of these languages is to separate the implementation of software artifacts, e.g., classes and methods, from the definition of features. That is, classes and refinements are not annotated or declared to belong to a feature. There is no statement in the program text that defines explicitly a connection between code and features. Instead, the mapping of software artifacts to features is established via containment hierarchies, as explained in Section 2. The advantage of this approach is that a feature's implementation can include, beside classes in the form of Java files, also other supporting documents, e.g., documentation in the form of HTML files, grammar specification in the form of JavaCC files, or build scripts and deployment descriptors in the form of XML files [10]. To this end, feature composition merges not only classes with their refinements but also other artifacts such as HTML or XML files with their respective refinements [2, 9].

Jiazzi and C# are two languages that are commonly not associated with FOP but that provide very similar mechanisms. With Jiazzi, a programmer can aggregate several classes in a component and compose them in a feature-oriented fashion [33]. The mapping between code and components is described externally by means of a separate linker language. In C#, there is the possibility to specify a class refinement, which is called a partial class. Aggregating a set of (partial) classes in a file system directory is very similar to feature-oriented languages, in which a feature's constituting ar-

tifacts are aggregated in a containment hierarchy. Thus, the results of studying FFJ provide insights in Jiazzi's and C#'s type system.

Another class of programming languages that provide mechanisms for the definition and extension of classes and class hierarchies includes, e.g., *ContextL* [20], *Scala* [38], and *Classbox/J* [11]. The difference to feature-oriented languages is that they provide explicit language constructs for aggregating the classes that belong to a feature, e.g., family classes, classboxes, or layers. This implies that noncode software artifacts cannot be included in a feature [8]. However, FFJ models still a subset of these languages, in particular, class refinement.

Similarly, related work on a formalization of the key concepts underlying FOP has not disassociated the concept of a feature from the level of code. Especially, calculi for mixins [18, 12, 1, 25], traits [30], family polymorphism and virtual classes [24, 17, 22, 13], path-dependent types [38, 37], open classes [14], dependent classes [19], and nested inheritance [36] either support only the refinement of single classes or expect the classes that form a semantically coherent unit (i.e., that belong to a feature) to be located in a physical module that is defined in the host programming language. For example, a virtual class is by definition an inner class of the enclosing object, and a classbox is a package that aggregates a set of related classes. Thus, FFJ differs from previous approaches in that it relies on contextual information that has been collected by the composition engine, e.g., the features' composition order or the mapping of code to features.

A different line of research aims at the language-independent reasoning about features [10, 31, 9, 29]. The calculus gDeep is most related to FFJ since it provides a type system for feature-oriented languages that is language-independent [3]. The idea is that the recursive process of merging software artifacts, when composing hierarchically structured features, is very similar for different host languages, e.g., for Java, C#, and XML. The calculus describes formally how feature composition is performed and what type constraints have to be satisfied. In contrast, FFJ does not aspire to be language-independent, although the key concepts can certainly be used with different languages. The advantage of FFJ is that its type system can be used to check whether terms of the host language (Java/FJ) violate the principles of FOP and stepwise refinement, e.g., whether methods refer to classes that have been added subsequently. Due to its language independence, gDeep does not have enough information to perform such checks; however, FFJ and gDeep could be combined.

Czarnecki et al. have presented an automatic verification procedure for ensuring that no ill-structured UML model template instances will be generated from a valid feature selection [16]. They use OCL (object constraint language) constraints to express and implement a type system for model composition. In this sense, their aim is very similar to FFJ, but limited to diagram artifacts.

Kästner et al. have implemented a tool, called CIDE, that allows a developer to refactor a legacy software system into features [28, 29]. In contrast to other feature-oriented languages and tools, the link between code and features is established via annotations. If a user selects a set of features, all code that is annotated with features that are not present in the selection is removed. Kästner et al. have developed a formal calculus and a set of type rules that ensure that only well-typed programs can be generated [27]. For example, if a method declaration is removed, the remaining code must not contain calls to this method. CIDE's type rules are similar to the type rules of FFJ. In some sense, our approach and the approach of CIDE are two sides of the same coin: one aims at feature composition and the other at feature decomposition.

6. Conclusion

Feature-oriented programming (FOP) is a paradigm that incorporates programming language technology, program generation techniques, and stepwise refinement. The question of what a type system for FOP should look like has not been answered before [43]. We have presented a type system for FOP on top of a simple, feature-oriented language, called FFJ. The type system can be used to check before compilation whether a given composition of features is safe. FFJ is interesting insofar as it incorporates reasoning at the programming language level and the composition engine at the meta level, which is different from previous work. We have been able to show that FFJ's type system is sound.

Furthermore, we have explored several variations of FFJ for stepwise refinement and have shown that they are type sound as well. FFJ is a promising start in experimenting with further extensions such as separate compilation, method signature extension, field overriding, feature interfaces, optional method refinements, and many more.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. The work was funded partly by the German Research Foundation (DFG), project number AP 206/2-1.

References

[1] D. Ancona, G. Lagorio, and E. Zucca. Jam—Designing a Java Extension with Mixins. *ACM Trans. Programming Languages and Systems (TOPLAS)*, 25(5):641–712, 2003.

[2] F. Anfurrutia, O. Díaz, and S. Trujillo. On Refining XML Artifacts. In *Proc. Int'l. Conf. Web Engineering (ICWE)*, volume 4607 of *LNCS*, pages 473–478. Springer-Verlag, 2007.

[3] S. Apel and D. Hutchins. An Overview of the gDeep Calculus. Technical Report MIP-0712, Department of Informatics and Mathematics, University of Passau, 2007.

[4] S. Apel, C. Kästner, and D. Batory. Program Refactoring using Functional Aspects. In *Proc. Int'l. Conf. Generative Programming and Component Engineering (GPCE)*. ACM Press, 2008.

[5] S. Apel, C. Kästner, T. Leich, and G. Saake. Aspect Refinement – Unifying AOP and Stepwise Refinement. *Journal of Object Technology – Special Issue: TOOLS EUROPE'07*, pages 13–33, 2007.

[6] S. Apel, C. Kästner, and C. Lengauer. An Overview of Feature Featherweight Java. Technical Report MIP-0802, Department of Informatics and Mathematics, University of Passau, 2008.

[7] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. Int'l. Conf. Generative Programming and Component Engineering (GPCE)*, volume 3676 of *LNCS*, pages 125–140. Springer-Verlag, 2005.

[8] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Trans. Software Engineering (TSE)*, 34(2):162–180, 2008.

[9] S. Apel and C. Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *Proc. Int'l. Symp. Software Composition (SC)*, volume 4954 of *LNCS*, pages 20–35. Springer-Verlag, 2008.

[10] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Engineering (TSE)*, 30(6):355–371, 2004.

[11] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 177–189. ACM Press, 2005.

[12] V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1628 of *LNCS*, pages 43–66. Springer-Verlag, 1999.

[13] D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad. Tribe: A Simple Virtual Class Calculus. In *Proc. Int'l. Conf. Aspect-Oriented Software Development (AOSD)*, pages 121–134. ACM Press, 2007.

[14] C. Clifton, T. Millstein, G. Leavens, and C. Chambers. MultiJava: Design Rationale, Compiler Implementation, and Applications. *ACM Trans. Programming Languages and Systems (TOPLAS)*, 28(3):517–575, 2006.

[15] W. Cook. Object-Oriented Programming Versus Abstract Data Types. In *Proc. REX School/Workshop Foundations of Object-Oriented Languages*, volume 489 of *LNCS*, pages 151–178. Springer-Verlag, 1991.

[16] K. Czarnecki and K. Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proc. Int'l. Conf. Generative Programming and Component Engineering (GPCE)*, pages 211–220. ACM Press, 2006.

[17] E. Ernst, K. Ostermann, and W. Cook. A Virtual Class Calculus. In *Proc. Int'l. Symp. Principles of Programming Languages (POPL)*, pages 270–282. ACM Press, 2006.

[18] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. In *Proc. Int'l. Symp. Principles of Programming Languages (POPL)*, pages 171–183. ACM Press, 1998.

[19] V. Gasiunas, M. Mezini, and K. Ostermann. Dependent Classes. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 133–152. ACM Press, 2007.

[20] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-Oriented Programming. *J. Object Technology (JOT)*, 7(3):125–151, 2008.

[21] S. Huang and Y. Smaragdakis. Expressive and Safe Static Reflection with MorphJ. In *Proc. Int'l. Conf. Programming Language Design and Implementation (PLDI)*, pages 79–89. ACM Press, 2008.

[22] D. Hutchins. Eliminating Distinctions of Class: Using Prototypes to Model Virtual Classes. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–19. ACM Press, 2006.

[23] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[24] A. Igarashi, C. Saito, and M. Viroli. Lightweight Family Polymorphism. In *Proc. Asian Symp. Programming Languages and Systems (APLAS)*, volume 3780 of *LNCS*, pages 161–177. Springer-Verlag, 2005.

[25] T. Kamina and T. Tamai. McJava – A Design and Implementation of Java with Mixin-Types. In *Proc. Asian Symp. Programming Languages and Systems (APLAS)*, volume 3302 of *LNCS*, pages 398–414. Springer-Verlag, 2004.

[26] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[27] C. Kästner and S. Apel. Type-checking Software Product Lines – A Formal Approach. In *Proc. Int'l. Conf. Automated Software Engineering (ASE)*. IEEE CS Press, 2008.

[28] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l. Conf. Software Engineering (ICSE)*. ACM Press, 2008.

[29] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Language-Independent Safe Decomposition of Legacy Applications into Features. Technical Report 02/2008, School of Computer Science, University of Magdeburg, 2008.

[30] L. Liquori and A. Spiwack. FeatherTrait: A Modest Extension of Featherweight Java. *ACM Trans. Programming Languages and Systems (TOPLAS)*, 30(2):1–32, 2008.

[31] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *Proc. Int'l. Symp. Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 68–77. ACM Press, 2006.

[32] O. Madsen, B. Moller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[33] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–222. ACM Press, 2001.

[34] S. McDirmid, W. Hsieh, and M. Flatt. A Framework for Modular Linking in OO Languages. In *Proc. Joint Modular Lang. Conf. (JMLC)*, volume 4228 of *LNCS*, pages 116–135. Springer-Verlag, 2006.

[35] N. McEachen and R. Alexander. Distributing Classes with Woven Concerns: An Exploration of Potential Fault Scenarios. In *Proc. Int'l. Conf. Aspect-Oriented Software Development (AOSD)*, pages 192–200. ACM Press, 2005.

[36] N. Nystrom, S. Chong, and A. Myers. Scalable Extensibility via Nested Inheritance. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 99–115. ACM Press, 2004.

[37] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A Nominal Theory of Objects with Dependent Types. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 2743 of *LNCS*, pages 201–224. Springer-Verlag, 2003.

[38] M. Odersky and M. Zenger. Scalable Component Abstractions. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 41–57. ACM Press, 2005.

[39] K. Ostermann. Nominal and Structural Subtyping in Component-Based Programming. *J. Object Technology (JOT)*, 7(1):121–145, 2008.

[40] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 419–443. Springer-Verlag, 1997.

[41] J. Reynolds. 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*, pages 13–23. 1994.

[42] A. Taivalsaari. On the Notion of Inheritance. *ACM Comp. Surv.*, 28(3):438–479, 1996.

[43] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proc. Int'l. Conf. Generative Programming and Component Engineering (GPCE)*, pages 95–104. ACM Press, 2007.

[44] M. Torgersen. The Expression Problem Revisited. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 123–143. Springer-Verlag, 2004.

[45] N. Wirth. Program Development by Stepwise Refinement. *Comm. ACM*, 14(4):221–227, 1971.

[46] A. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Inf. Comput.*, 115(1):38–94, 1994.