# Type Safety for Feature-Oriented Product Lines

**Sven Apel · Christian Kästner · Armin
Größlinger · Christian Lengauer**

**Abstract** A *feature-oriented product line* is a family of programs that share a common set of features. A *feature* implements a stakeholder's requirement and represents a design decision or configuration option. When added to a program, a feature involves the introduction of new structures, such as classes and methods, and the refinement of existing ones, such as extending methods. A feature-oriented decomposition enables a generator to create an executable program by composing feature code solely on the basis of the feature selection of a user – no other information needed. A key challenge of product line engineering is to guarantee that only well-typed programs are generated. As the number of valid feature combinations grows combinatorially with the number of features, it is not feasible to type check all programs individually. The only feasible approach is to have a type system check the entire code base of the feature-oriented product line. We have developed such a type system on the basis of a formal model of a feature-oriented Java-like language. The type system guaranties type safety for feature-oriented product lines. That is, it ensures that *every* valid program of a well-typed product line is well-typed. Our formal model including type system is sound and complete.

## 1 Introduction

*Feature-oriented programming* (*FOP*) aims at the modularization of programs in terms of features [59, 15]. A *feature* implements a stakeholder's requirement and represents a de-

Sven Apel
Department of Informatics and Mathematics, University of Passau, Germany
E-mail: apel@uni-passau.de

Christian Kästner
School of Computer Science, University of Magdeburg, Germany
E-mail: kaestner@iti.cs.uni-magdeburg.de

Armin Größlinger
Department of Informatics and Mathematics, University of Passau, Germany
E-mail: groesslinger@uni-passau.de

Christian Lengauer
Department of Informatics and Mathematics, University of Passau, Germany
E-mail: lengauer@uni-passau.de

sign decision or configuration option [7]. Contemporary feature-oriented programming languages and tools, such as AHEAD [15], Xak [2], CaesarJ [52], Classbox/J [16], Feature-House [10], and FeatureC++ [11], provide a variety of mechanisms that support the specification, modularization, and composition of features. A key idea is that a feature is implemented by a distinct code unit, called a *feature module*. When added to a base program, it introduces new structures, such as classes and methods, and refines existing ones, such as extending methods [47,12]. A program that is decomposed into features is called henceforth a *feature-oriented program.*[1]

Beside the decomposition of programs into feature modules, the concept of a feature is useful for distinguishing different, related programs which together make up a *software product line* [37,23]. Typically, programs of a common domain share a set of features but also differ in other features. For example, suppose an email client for mobile devices that supports the protocols IMAP and POP3 and another client that supports POP3, MIME, and SSL encryption. With a decomposition of the two programs into the features IMAP, POP3, MIME, and SSL, both programs can share the code of feature POP3. Since mobile devices have only limited resources, unnecessary features should be removed.

With feature-oriented decomposition, programs can be generated solely on the basis of a user's selection of features by composing the corresponding feature modules. Of course, not all combinations of features are legal and result in correct programs [14]. A *feature model* describes which features can be composed in which combinations, i.e., which programs are *valid* [37,23]. It consists of an (ordered) set of features and a set of constraints on feature combinations [23,14]. For example, our email client may have different rendering engines for HTML text, e.g., the Mozilla engine or the Safari engine, but only one at a time. A set of feature modules along with a feature model is called a *feature-oriented product line* [14].

An important question is how the correctness of feature-oriented programs, in particular, and product lines, in general, can be guaranteed. A first problem is that contemporary feature-oriented languages and tools usually involve a code generation step during composition in which the code is transformed into a lower-level representation. In previous work, we have addressed this problem by modeling feature-oriented mechanisms directly in the formal syntax and semantics of a core language, called *Feature Featherweight Java* (*FFJ*) [9]. The type system of FFJ ensures that the composition of feature modules is type-correct.

Here, we address a second problem: How can the type safety of a feature-oriented product line be guaranteed? That is, are all valid programs of the product line well-typed? A naive approach would be to type check all valid programs using a type checker that expects single programs, like the one for FFJ [9]. However, this approach does not scale; already for 34 independent optional features, a variant can be generated for every person on the planet. Noticing this problem, Czarnecki and Pietroszek [24] and Thaker et al. [69] suggested the development of a type system that checks the entire code and document base of the feature-oriented product line, instead of all individual feature-oriented programs. In this scenario, a type checker must analyze *all* feature modules of a product line on the basis of the feature model. We will show that, with this information, the type checker can ensure type safety. That is, *every* valid program variant that can be generated is well-typed. Specifically, we make the following contributions:

---

[1] Typically, feature-oriented decomposition is orthogonal to class-based or functional decomposition [68, 53,66]. A multitude of modularization and composition mechanisms [19,28,26,49,50,60,70] have been developed to allow programmers to decompose a program along multiple dimensions [68]. Feature-oriented languages and tools provide a significant subset of these mechanisms [12].

– We provide a condensed version of FFJ, which is in many respects more elegant and concise than its predecessor [9].
– We develop a formal type system that uses information about features and constraints on feature combinations to type check an entire product line without generating every program.
– We show that the type system is *sound* (i.e., it guarantees that every program generated from a well-typed product line is well-typed). Furthermore, we show that the type system is *complete* (i.e., the well-typedness of all programs of a product line implies that the product line is well-typed as a whole).
– We offer an implementation of FFJ, including the proposed type system, which can be downloaded for evaluation and for experiments with further feature-oriented language and typing mechanisms.

Our work differs in many respects from previous and related work (see Section 5 for a comprehensive discussion). Most notably, Thaker et al. have implemented a type system for feature-oriented product lines and conducted several case studies [69]. We take their work further with a formalization and a soundness and completeness proof. A further distinguishing property is that we model feature-related mechanisms directly in FFJ's semantics and type system, without any transformation to a lower-level representation (e.g., as in the work of Delaware et al. [25]), and we stay very close to the syntax of contemporary feature-oriented languages and tools. Finally, our work is related to type-checking mechanisms for annotation-based product lines [38]. The type systems of traditional feature-oriented product lines and annotation-based product lines are complementary to some extent. However, our approach supports the full power of alternative features including all implications such as that terms may have multiple types. For example, previous work by Kästner et al. supports only one type per term.

## 2 Feature-Oriented Programs in FFJ

In this section, we introduce the language FFJ. Originally, FFJ was designed for feature-oriented programs [9]. We extend FFJ in Section 3 to support feature-oriented product lines, i.e., to support the representation of multiple alternative program variants at a time.

### 2.1 An Overview of FFJ

FFJ is a lightweight feature-oriented language that has been inspired by *Featherweight Java* (*FJ*) [34]. As with FJ, we have aimed at minimality in the design of FFJ. FFJ provides basic constructs like classes, fields, methods, and inheritance and only a few new constructs capturing the core mechanisms of FOP. But, so far, FFJ's type system has not supported the development of feature-oriented product lines. That is, a set of feature modules written in FFJ constitutes a single program. We will change this in Section 3.

Based on an earlier version [9], we developed a condensed version of FFJ, which is in many respects more elegant and concise than its predecessor. After explaining the condensed version, we summarize briefly the differences to the earlier version in Section 2.9.

An FFJ program consists of a set of classes and refinements. A *refinement* extends a class that has been introduced previously. Each class and refinement is associated with a

feature. We say that a feature *introduces* a class or *applies* a refinement to a class. Technically, the mapping between classes/refinements and the feature they belong to can be established in different ways, e.g., by extending the language with modules representing features [52, 16, 25] or by grouping classes and refinements that belong to a feature in packages or directories [15, 11]. In the remainder, we call the set of classes and refinements associated with a feature a feature module, neglecting that modules are not explicit in FFJ.

Like in FJ, each class declares a superclass, which may be the class Object. Refinements are defined using the keyword refines. The semantics of a refinement applied to a class is that the refinement's members are added to and merged with the members of the refined class. This way, a refinement can *add* new fields and methods to the class and *override* existing methods (declared by modifier overrides).

On the left side in Figure 1, we show an excerpt of the FFJ code of a basic email client called EMAILCLIENT (top) and a feature called SSL (bottom). Feature SSL adds class SSL (Lines 7–10) to the email client's code base and refines class Trans to encrypt outgoing messages (Lines 11–15). To this effect, the refinement of Trans adds a new field key (Line 12) and overrides the method send of class Trans (Lines 13-15).
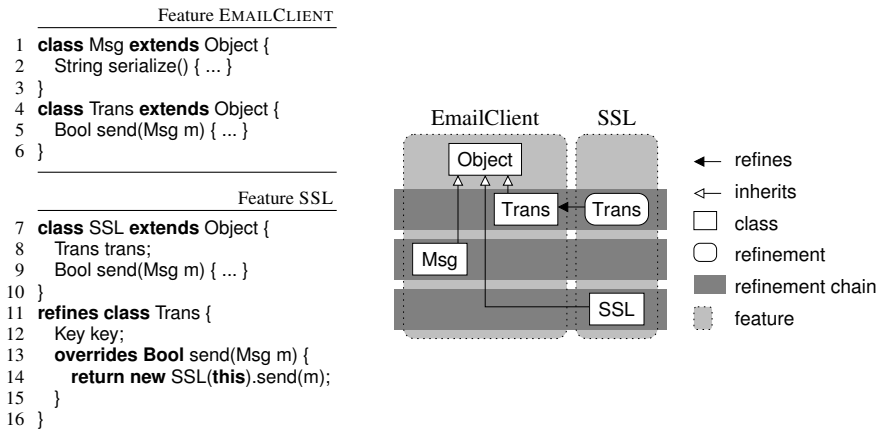


**Fig. 1** A feature-oriented email client supporting SSL encryption.

Typically, a programmer applies multiple refinements to a class by composing a sequence of features. The ordered list of refinements is called a *refinement chain*. A refinement that is applied immediately before another refinement in the chain is called its *predecessor*. The order of the refinements in a refinement chain is determined by their features' composition order. On the right side in Figure 1, we depict the refinement and inheritance relationships of our email example.

Fields are unique within the scope of a class, its inheritance hierarchy, and its refinement chain (i.e., field overshadowing is prohibited, for simplicity). That is, a refinement or subclass is not allowed to add a field that has already been defined in a predecessor in the refinement chain or in a superclass. For example, a further refinement of Trans would not be allowed to add a field key, since key has been introduced by feature SSL already. With methods, this is different. A refinement or subclass may add new methods (overloading is prohibited, for simplicity) *and* override existing methods. To distinguish the two cases, FFJ

expects the programmer to declare whether a method overrides an existing method (using modifier overrides). For example, the refinement of Trans in feature SSL overrides the method send introduced by feature EMAILCLIENT; for subclasses, this is similar.

The distinction between method introduction and overriding allows the type system to check (1) whether an introduced method inadvertently replaces or occludes an existing method with the same name and (2) whether, for every overriding method, there is a proper method to be overridden. Apart from the modifier overrides, the syntax of methods in FFJ is identical to the syntax methods in FJ. That is, a method 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 (introducing methods with equal names and different argument types) is not allowed in FFJ (and FJ).

As shown in Figure 1, 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 a class and selects the right-most and bottom-most (i.e., the least in the lexical order depicted in Figure 1) body of a method declaration or method refinement that is compatible. This kind of lookup is necessary since we model features *directly* in FFJ, instead of generating and evaluating FJ code [44].

First, FFJ's method lookup mechanism searches for a proper method declaration in the refinement chain of the given class, starting with the last refinement back to the class declaration itself. The first body of a matching method declaration is returned. If the method is not found in the class' refinement chain or in its own declaration, the methods in the superclass (and then the superclass' superclass, etc.) are searched, each again from the most specific refinement to the class declaration itself. The field lookup works analogously, except that the entire inheritance and refinement hierarchy is searched and the fields are accumulated in a list. In Figure 2, we illustrate the processes of method body and field lookup schematically.
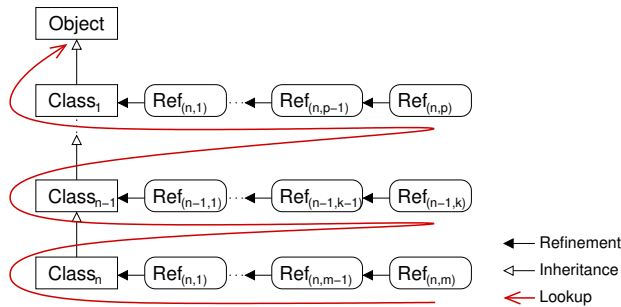


**Fig. 2** Order of method body and field lookup in FFJ.

2.2 Syntax of FFJ

Before we go into detail, let us introduce some notational conventions. We abbreviate lists in the obvious ways:

– $\overline{\mathsf{C}}$ is shorthand for $\mathsf{C}_1, \ldots, \mathsf{C}_n$
– $\overline{\mathsf{C}\,\mathsf{f}}$ is shorthand for $\mathsf{C}_1\,\mathsf{f}_1, \ldots, \mathsf{C}_n\,\mathsf{f}_n$

| L ::= | *class declarations:* |
|---|---|
| class C extends D { $\overline{C\ f}$; $\overline{M}$ } | |
| R ::= | *refinement declarations:* |
| refines class C { $\overline{C\ f}$; $\overline{M}$ } | |
| M ::= | *method declarations:* |
| [overrides] C m($\overline{C\ x}$) { return t; } | |

| t ::= | *terms:* |
|---|---|
| x | *variable* |
| t.f | *field access* |
| t.m($\overline{t}$) | *method invocation* |
| new C($\overline{t}$) | *object creation* |
| (C) t | *cast* |
| v ::= | *values:* |
| new C($\overline{v}$) | *object creation* |

**Fig. 3** Syntax of FFJ in extended BNF.

- $\overline{C\ f}$; is shorthand for $C_1\ f_1$; ... ; $C_n\ f_n$;
- $\overline{t : C}$ is shorthand for $t_1 : C_1$, ... , $t_n : C_n$
- $\overline{C <: D}$ is shorthand for $C_1 <: D_1\ ...\ C_n <: D_n$
- ...

Note that, depending on the context, blanks, commas, or semicolons separate the elements of a list. The context will make clear which separator is meant. The symbol • denotes the empty list, and lists of field declarations, method declarations, and parameter names must not contain duplicates (by name). We use the metavariables A–E for class names, f–h for field names, and m for method names. Feature names are denoted by Greek letters.

In Figure 3, we depict the syntax of FFJ in extended Backus-Naur-Form. An FFJ program consists of a set of class and refinement declarations. A class declaration L declares a class with the name C that inherits from a superclass D and consists of a list $\overline{C\ f}$; of fields and a list $\overline{M}$ of method declarations.[2] A refinement declaration R consists of a list $\overline{C\ f}$; of fields and a list $\overline{M}$ of method declarations.

A method m expects a list $\overline{C\ x}$ of arguments and declares a body that returns only a single expression t of type C. Using the modifier overrides, a method declares that it intends to override another method with an equal signature. Where we want to distinguish methods that override others and methods that do not override others, we call the former *method introductions* and the latter *method refinements*

Finally, there are five forms of terms: the variable, field access, method invocation, object creation, and type cast, which are taken from FJ without change. The only values are object creations whose arguments are values as well.

### 2.3 FFJ's Class Table

Declarations of classes and refinements can be looked up via a class table *CT*. The compiler fills the class table during the parser pass. In contrast to FJ, class and refinement declarations are identified not only by their names but, additionally, by the names of the enclosing features. For example, in order to retrieve the declaration of class Trans, introduced by feature EMAILCLIENT, in our example of Figure 1, we write *CT*(EMAILCLIENT.Trans); in order to retrieve the refinement of class Trans applied by feature SSL, we write *CT*(SSL.Trans). We call Φ.C the *qualified type* of class C in feature Φ. The reserved symbol Base denotes the feature that introduces class Object. Feature Base plays the same role for features as Object plays for classes.

---

[2] The concept of a class constructor is unnecessary in FFJ and FJ [58]. In FJ, it is used solely for backward compatibility with Java. In FFJ, we do not pursue backward compatibility; its omittance simplifies the syntax, semantics, and type rules significantly without loss of generality.

*Navigating along the refinement chain*

$$\frac{RT(\mathsf{C}) = \overline{\Psi}}{last(\mathsf{C}) = \Psi_n.\mathsf{C}} \qquad \frac{RT(\mathsf{C}) = \overline{\Psi}, \Phi, \overline{\Omega} \qquad \overline{\Psi} \neq \bullet}{pred(\Phi.\mathsf{C}) = \Psi_n.\mathsf{C}} \qquad \frac{RT(\mathsf{C}) = \Phi, \overline{\Omega}}{pred(\Phi.\mathsf{C}) = \mathrm{Base}.\mathsf{Object}}$$

**Fig. 4** Refinement in FFJ.

In FFJ, class and refinement declarations are unique with respect to their qualified types. Our model ensures this property by the following sanity conditions: a feature is not allowed

- to introduce a class or refinement twice inside a single feature module and
- to refine a class that the feature has just introduced.

These are common sanity conditions in feature-oriented languages and tools [15, 11, 10]. As for FJ, we impose further sanity conditions on the class table and the inheritance relation:

- $CT(\Phi.\mathsf{C}) =$ class C extends D { ... } or refines class C { ... } for every qualified type $\Phi.\mathsf{C} \in dom(CT)$;
- $\Phi.\mathsf{C} \in dom(CT) \Rightarrow \Phi \neq \mathrm{Base} \wedge \mathsf{C} \neq \mathsf{Object}$;
- for every class name C appearing anywhere in $CT$, we have $\Phi.\mathsf{C} \in dom(CT)$ for at least one feature $\Phi$; and
- the inheritance relation contains no cycles (including self-cycles).

### 2.4 Refinement in FFJ

Information on the refinement chain of a class can be retrieved using the refinement table $RT$. The compiler fills the refinement table during the parser pass. It can be constructed solely on the basis of the class table. $RT(\mathsf{C})$ yields a list of all features that either introduce or refine class C. Specifically, $RT(\mathsf{C}) = \overline{\Phi}$ for every type $\Psi.\mathsf{C} \in dom(CT)$, with $\Phi_1$ being the feature that introduces class C (i.e., $\overline{\Phi} \neq \bullet$) and a possibly empty list $\Phi_2, \ldots \Phi_\mathrm{n}$ of features that refine class C. That is, the leftmost element of the result list is the feature that introduces the class C and, then, from left to right, the features are listed that refine class C in the order of their composition. In our example of Figure 1, $RT(\mathsf{Trans})$ yields the list EMAILCLIENT, SSL.

In Figure 4, we show two functions for the navigation of the refinement chain that rely on $RT$. Function *last* returns, for a class name C, a qualified type $\Psi_n.\mathsf{C}$, in which $\Psi_n$ refers to the feature that applies the final refinement to class C; if a class is not refined at all, $\Psi_n$ refers to the feature that introduces class C. Function *pred* returns, for a qualified type $\Phi.\mathsf{C}$, another qualified type $\Psi_n.\mathsf{C}$, in which $\Psi_n$ refers to the feature that introduces or refines class C and that is the immediate predecessor of $\Phi$ in the refinement chain; if there is no predecessor, $\mathrm{Base}.\mathsf{Object}$ is returned (for terminating the lookup functions).

### 2.5 Subtyping in FFJ

In Figure 5, we show the subtype relation <: of FFJ. It is defined by one rule for reflexivity, one rule for transitivity, and one rule for relating the type of a class to the type of its immediate superclass. It is not necessary to define subtyping over qualified types because only classes (not refinements) declare superclasses and there is only a single declaration per class.

*Subtyping*

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

$$C <: C \qquad \frac{C <: D \quad D <: E}{C <: E} \qquad \frac{CT(\Phi.C) = \text{class C extends D } \{\dots\}}{C <: D}$$

**Fig. 5** Subtyping in FFJ.

### 2.6 Auxiliary Definitions of FFJ

In Figure 6, we show the auxiliary definitions of FFJ. Function *fields* searches the refinement chain from right to left and accumulates the fields into a list (using the comma as concatenation operator). If there is no further predecessor in the refinement chain (i.e., we have reached a class declaration), then the refinement chain of the superclass is searched (see Figure 2). If $\text{Base.Object}$ is reached, the empty list is returned (denoted by •).

Function *mbody* looks up the most specific and most refined body of a method m. A body consists of the formal parameters $\bar{x}$ of a method and a term t representing the content. The search is like in *fields*. First, the refinement chain is searched from right to left and, then, the superclasses' refinement chains are searched, as illustrated in Figure 2. Note that [overrides] means that a given method declaration may (or may not) have the modifier. This way, we are able to define uniform rules for method introduction and method refinement.

Function *mtype* yields the signature $\bar{B} \rightarrow B_0$ of a declaration of method m. The lookup is like in *mbody*, except that method introductions are considered only. Later we define a well-formedness rule that guarantees that all corresponding method refinements have compatible types.

Predicate $introduce_{class}$ is used to check whether a class has been introduced by multiple features. Precisely, it states whether C has not been introduced by any feature other than $\Phi$. To evaluate it, we check whether $CT(\Psi.C)$ yields a class declaration or not, for any feature $\Psi$ different from $\Phi$. Similarly, $introduce_{field}$ and $introduce_{method}$ are used to check whether a field f or a method m has been introduced multiple times in a class (i.e., whether they have not been introduced by $\Phi.C$ or in any of its predecessors or superclasses). In the case of methods, we check whether $mtype$ yields a signature and, in the case of fields, we check whether f is defined in the list of fields returned by *fields*.

Predicate *refine* states whether, for a given refinement, a proper class has been declared previously in the refinement chain. The predicate *override* states whether a method m has been introduced before in some predecessor of $\Phi.C$ and whether the previous declaration of m has the given signature.

### 2.7 Evaluation of FFJ Programs

Each FFJ program consists of a class table, a refinement table, and a term. The term is evaluated using the evaluation rules shown in Figure 7. The evaluation terminates when a value (i.e., a term of the form $\text{new C}(\bar{v})$) is reached. Note that we use a *direct semantics* of class refinement [44]. That is, the field and method lookup mechanisms incorporate all refinements when a class is searched for fields and methods. An alternative would be a *flattening semantics* that merges each class in a preprocessing step with all of its refinements into a single declaration. In Section 5, we compare both alternatives and justify our choice.

Using the subtype relation $<:$ and the auxiliary functions *fields* and *mbody*, the evaluation of FFJ is fairly simple. The first three rules are most interesting (the remaining rules are

*Field lookup*  $\boxed{\textit{fields}(\Phi.\mathsf{C}) = \overline{\mathsf{C}\,\mathsf{f}}}$

$$\textit{fields}(\mathrm{Base}.\mathsf{Object}) = \bullet$$

$$\frac{CT(\Phi.\mathsf{C}) = \text{class C extends D} \{ \overline{\mathsf{C}\,\mathsf{f}};\ \overline{\mathsf{M}} \}}{\textit{fields}(\Phi.\mathsf{C}) = \textit{fields}(\textit{last}(\mathsf{D})),\ \overline{\mathsf{C}\,\mathsf{f}}} \qquad \frac{CT(\Phi.\mathsf{C}) = \text{refines class C} \{ \overline{\mathsf{C}\,\mathsf{f}};\ \overline{\mathsf{M}} \}}{\textit{fields}(\Phi.\mathsf{C}) = \textit{fields}(\textit{pred}(\Phi.\mathsf{C})),\ \overline{\mathsf{C}\,\mathsf{f}}}$$

*Method body lookup*  $\boxed{\textit{mbody}(\mathsf{m}, \Phi.\mathsf{C}) = (\overline{\mathsf{x}}, \mathsf{t})}$

$$\frac{\begin{array}{c}CT(\Phi.\mathsf{C}) = \text{class C extends D} \{ \overline{\mathsf{C}\,\mathsf{f}};\ \overline{\mathsf{M}} \}\\ [\text{overrides}]\ \mathsf{B}\ \mathsf{m}(\overline{\mathsf{B}\,\mathsf{x}})\ \{ \text{return t; } \} \in \overline{\mathsf{M}}\end{array}}{\textit{mbody}(\mathsf{m}, \Phi.\mathsf{C}) = (\overline{\mathsf{x}}, \mathsf{t})} \qquad \frac{\begin{array}{c}CT(\Phi.\mathsf{C}) = \text{class C extends D} \{ \overline{\mathsf{C}\,\mathsf{f}};\ \overline{\mathsf{M}} \}\\ \mathsf{m}\ \text{is not defined in}\ \overline{\mathsf{M}}\end{array}}{\textit{mbody}(\mathsf{m}, \Phi.\mathsf{C}) = \textit{mbody}(\mathsf{m}, \textit{last}(\mathsf{D}))}$$

$$\frac{\begin{array}{c}CT(\Phi.\mathsf{C}) = \text{refines class C} \{ \overline{\mathsf{C}\,\mathsf{f}};\ \overline{\mathsf{M}} \}\\ [\text{overrides}]\ \mathsf{B}\ \mathsf{m}(\overline{\mathsf{B}\,\mathsf{x}})\ \{ \text{return t; } \} \in \overline{\mathsf{M}}\end{array}}{\textit{mbody}(\mathsf{m}, \Phi.\mathsf{C}) = (\overline{\mathsf{x}}, \mathsf{t})} \qquad \frac{\begin{array}{c}CT(\Phi.\mathsf{C}) = \text{refines class C} \{ \overline{\mathsf{C}\,\mathsf{f}};\ \overline{\mathsf{M}} \}\\ \mathsf{m}\ \text{is not defined in}\ \overline{\mathsf{M}}\end{array}}{\textit{mbody}(\mathsf{m}, \Phi.\mathsf{C}) = \textit{mbody}(\mathsf{m}, \textit{pred}(\Phi.\mathsf{C}))}$$

*Method type lookup*  $\boxed{\textit{mtype}(\mathsf{m}, \Phi.\mathsf{C}) = \overline{\mathsf{C}} \rightarrow \mathsf{C}}$

$$\frac{\begin{array}{c}CT(\Phi.\mathsf{C}) = \text{class C extends D} \{ \overline{\mathsf{C}\,\mathsf{f}};\ \overline{\mathsf{M}} \}\\ \mathsf{B}_0\ \mathsf{m}(\overline{\mathsf{B}\,\mathsf{x}})\ \{ \text{return t; } \} \in \overline{\mathsf{M}}\end{array}}{\textit{mtype}(\mathsf{m}, \Phi.\mathsf{C}) = \overline{\mathsf{B}} \rightarrow \mathsf{B}_0} \qquad \frac{\begin{array}{c}CT(\Phi.\mathsf{C}) = \text{class C extends D} \{ \overline{\mathsf{C}\,\mathsf{f}};\ \overline{\mathsf{M}} \}\\ \mathsf{m}\ \text{is not defined in}\ \overline{\mathsf{M}}\end{array}}{\textit{mtype}(\mathsf{m}, \Phi.\mathsf{C}) = \textit{mtype}(\mathsf{m}, \textit{last}(\mathsf{D}))}$$

$$\frac{\begin{array}{c}CT(\Phi.\mathsf{C}) = \text{refines class C} \{ \overline{\mathsf{C}\,\mathsf{f}};\ \overline{\mathsf{M}} \}\\ \mathsf{B}_0\ \mathsf{m}(\overline{\mathsf{B}\,\mathsf{x}})\ \{ \text{return t; } \} \in \overline{\mathsf{M}}\end{array}}{\textit{mtype}(\mathsf{m}, \Phi.\mathsf{C}) = \overline{\mathsf{B}} \rightarrow \mathsf{B}_0} \qquad \frac{\begin{array}{c}CT(\Phi.\mathsf{C}) = \text{refines class C} \{ \overline{\mathsf{C}\,\mathsf{f}};\ \overline{\mathsf{M}} \}\\ \mathsf{m}\ \text{is not defined in}\ \overline{\mathsf{M}}\end{array}}{\textit{mtype}(\mathsf{m}, \Phi.\mathsf{C}) = \textit{mtype}(\mathsf{m}, \textit{pred}(\Phi.\mathsf{C}))}$$

*Valid class introduction*  $\boxed{\textit{introduce}_{class}(\Phi.\mathsf{C})}$

$$\frac{\nexists\ \Psi\ :\ (CT(\Psi.\mathsf{C}) = \text{class C extends D} \{ \ldots \}\ \wedge\ \Phi \neq \Psi)}{\textit{introduce}_{class}(\Phi.\mathsf{C})}$$

*Valid field introduction*  $\boxed{\textit{introduce}_{field}(\mathsf{f}, \Phi.\mathsf{C})}$

$$\frac{\textit{fields}(\Phi.\mathsf{C}) = \overline{\mathsf{E}\,\mathsf{h}} \qquad \mathsf{f} \notin \overline{\mathsf{h}}}{\textit{introduce}_{field}(\mathsf{f}, \Phi.\mathsf{C})}$$

*Valid method introduction*  $\boxed{\textit{introduce}_{method}(\mathsf{m}, \Phi.\mathsf{C})}$

$$\frac{(\mathsf{m}, \Phi.\mathsf{C}) \notin \textit{dom}(\textit{mtype})}{\textit{introduce}_{method}(\mathsf{m}, \Phi.\mathsf{C})}$$

*Valid class refinement*  $\boxed{\textit{refine}(\Phi.\mathsf{C})}$

$$\frac{RT(\mathsf{C}) = \overline{\Psi}, \Phi, \overline{\Omega} \qquad CT(\Psi_1.\mathsf{C}) = \text{class C extends D} \{ \ldots \}}{\textit{refine}(\Phi.\mathsf{C})}$$

*Valid method overriding*  $\boxed{\textit{override}(\mathsf{m}, \Phi.\mathsf{C}, \overline{\mathsf{C}} \rightarrow \mathsf{C}_0)}$

$$\frac{\textit{mtype}(\mathsf{m}, \Phi.\mathsf{C}) = \overline{\mathsf{B}} \rightarrow \mathsf{B}_0 \qquad \overline{\mathsf{C}} = \overline{\mathsf{B}} \qquad \mathsf{C}_0 = \mathsf{B}_0}{\textit{override}(\mathsf{m}, \Phi.\mathsf{C}, \overline{\mathsf{C}} \rightarrow \mathsf{C}_0)}$$

**Fig. 6** Auxiliary definitions of FFJ.

$$\frac{\textit{fields}(\textit{last}(\mathsf{C})) = \overline{\mathsf{C}}\ \mathsf{f}}{(\mathsf{new}\ \mathsf{C}(\overline{\mathsf{v}})).\mathsf{f}_i \ \longrightarrow\ \mathsf{v}_i} \qquad (\text{E-PROJNEW})$$

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

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

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

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

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

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

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

**Fig. 7** Evaluation of FFJ programs.

congruence rules). Rule E-PROJNEW describes the projection of a field from an instantiated class. A projected field $\mathsf{f}_i$ evaluates to a value $\mathsf{v}_i$ that has been passed as argument to the instantiation. Function *fields* is used to look up the fields of the given class. It receives $\textit{last}(\mathsf{C})$ as argument since we want to search the entire refinement chain of class $\mathsf{C}$ from right to left (cf. Figure 2).

Rule E-PROJINVK evaluates a method invocation by replacing the invocation with the method's body. The formal parameters of the method are substituted in the body for the arguments of the invocation; the value on which the method is invoked is substituted for this. The function *mbody* is called with the last refinement of the class $\mathsf{C}$ in order to search the refinement chain from right to left and return the most specific method body (cf. Figure 2).

Rule E-CASTNEW evaluates an upcast by simply removing the cast. The premise must be that the cast is really an upcast and not a downcast or an incorrect cast.

2.8 Type Checking FFJ Programs

The type relation of FFJ consists of the type rules for terms and the well-formedness rules for classes, refinements, and methods, shown in Figures 8 and 9.

*2.8.1 Term Typing Rules.*

A term typing judgment is a triple consisting of a typing context $\Gamma$, a term $\mathsf{t}$, and a type $\mathsf{C}$ (see Figure 8).

Rule T-VAR checks whether a free variable is contained in the typing context. Rule T-FIELD checks whether a field access $\mathsf{t}_0.\mathsf{f}$ is well-typed. Specifically, it checks whether $\mathsf{f}$ is

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

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

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

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

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

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

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

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

**Fig. 8** Term typing in FFJ.

declared in the type of $\mathsf{t}_0$ and whether the type of $\mathsf{f}$ equals the type of the entire term. Rule T-INVK checks whether a method invocation $\mathsf{t}_0.\mathsf{m}(\overline{\mathsf{t}})$ is well-typed. To this end, it checks whether the arguments $\overline{\mathsf{t}}$ of the invocation are subtypes of the types of the formal parameters of $\mathsf{m}$ and whether the return type of $\mathsf{m}$ equals the type of the entire term. Rule T-NEW checks whether an object creation $\mathsf{new}\ \mathsf{C}(\overline{\mathsf{t}})$ is well-typed in that it checks whether the arguments $\overline{\mathsf{t}}$ of the instantiation of $\mathsf{C}$ are subtypes of the types $\overline{\mathsf{D}}$ of the fields of $\mathsf{C}$ and whether $\mathsf{C}$ equals the type of the entire term. The rules T-UCAST, T-DCAST, and T-SCAST check whether casts are well-typed. In each rule, it is checked whether the type $\mathsf{C}$ the term $\mathsf{t}_0$ is cast to is a subtype, supertype, or unrelated type of the type of $\mathsf{t}_0$ and whether $\mathsf{C}$ equals the type of the entire term.[3]

### 2.8.2 Well-Formedness Rules.

In Figure 9, we show FFJ's well-formedness rules of classes, refinements, and methods.

The well-formedness judgments of classes and refinements are binary relations between a class or refinement declaration and a feature, written $\mathsf{L}$ OK $\dashv \Phi$ and $\mathsf{R}$ OK $\dashv \Phi$. The rule of classes checks whether there is no feature other than $\Phi$ that introduces a class $\mathsf{C}$, whether none of the fields of the class declaration is introduced multiple times in the combined inheritance and refinement hierarchy, and whether all methods are well-formed in the context of the class' qualified type. The well-formedness rule of refinements is analogous, except that the rule checks whether a corresponding class has been introduced before.

---

[3] Rule T-SCAST is needed only for the small step semantics of FFJ (and FJ) to be able to formulate and prove the type preservation property. FFJ (and FJ) programs whose type derivation contains this rule (i.e., the premise *stupid warning* appears in the derivation) are not further considered (cf. [34]).

*Method typing* $\boxed{\text{M OK} \dashv \Phi.\text{C}}$

$$\frac{\overline{\text{x}:\text{B}}, \text{this}:\text{C} \vdash \text{t}_0:\text{E}_0 \qquad \text{E}_0 <: \text{B}_0}{CT(\Phi.\text{C}) = \text{class C extends D} \{\, \overline{\text{C f}}; \overline{\text{M}} \,\} \qquad introduce_{method}(\text{m}, last(\text{D}))}$$
$$\overline{\text{B}_0 \, \text{m}(\overline{\text{B x}}) \, \{ \, \text{return t}_0; \, \} \, \text{OK} \dashv \Phi.\text{C}}$$

$$\frac{\overline{\text{x}:\text{B}}, \text{this}:\text{C} \vdash \text{t}_0:\text{E}_0 \qquad \text{E}_0 <: \text{B}_0}{CT(\Phi.\text{C}) = \text{class C extends D} \{\, \overline{\text{C f}}; \overline{\text{M}} \,\} \qquad override(\text{m}, last(\text{D}), \overline{\text{B}} \rightarrow \text{B}_0)}$$
$$\overline{\text{overrides B}_0 \, \text{m}(\overline{\text{B x}}) \, \{ \, \text{return t}_0; \, \} \, \text{OK} \dashv \Phi.\text{C}}$$

$$\frac{\overline{\text{x}:\text{B}}, \text{this}:\text{C} \vdash \text{t}_0:\text{E}_0 \qquad \text{E}_0 <: \text{B}_0}{CT(\Phi.\text{C}) = \text{refines class C} \{\, \overline{\text{C f}}; \overline{\text{M}} \,\} \qquad introduce_{method}(\text{m}, pred(\Phi.\text{C}))}$$
$$\overline{\text{B}_0 \, \text{m}(\overline{\text{B x}}) \, \{ \, \text{return t}_0; \, \} \, \text{OK} \dashv \Phi.\text{C}}$$

$$\frac{\overline{\text{x}:\text{B}}, \text{this}:\text{C} \vdash \text{t}_0:\text{E}_0 \qquad \text{E}_0 <: \text{B}_0}{CT(\Phi.\text{C}) = \text{refines class C} \{\, \overline{\text{C f}}; \overline{\text{M}} \,\} \qquad override(\text{m}, pred(\Phi.\text{C}), \overline{\text{B}} \rightarrow \text{B}_0)}$$
$$\overline{\text{overrides B}_0 \, \text{m}(\overline{\text{B x}}) \, \{ \, \text{return t}_0; \, \} \, \text{OK} \dashv \Phi.\text{C}}$$

*Class typing* $\boxed{\text{L OK} \dashv \Phi}$

$$\frac{introduce_{class}(\Phi.\text{C}) \qquad \forall \text{f} \in \overline{\text{f}} : introduce_{field}(\text{f}, last(\text{D})) \qquad \overline{\text{M}} \, \text{OK} \dashv \Phi.\text{C}}{\text{class C extends D} \{\, \overline{\text{C f}}; \overline{\text{M}} \,\} \, \text{OK} \dashv \Phi}$$

*Refinement typing* $\boxed{\text{R OK} \dashv \Phi}$

$$\frac{refine(\Phi.\text{C}) \qquad \forall \text{f} \in \overline{\text{f}} : introduce_{field}(\text{f}, pred(\Phi.\text{C})) \qquad \overline{\text{M}} \, \text{OK} \dashv \Phi.\text{C}}{\text{refines class C} \{\, \overline{\text{C f}}; \overline{\text{M}} \,\} \, \text{OK} \dashv \Phi}$$

**Fig. 9** Well-formedness rules of FFJ.

The well-formedness judgment of methods is a binary relation between a method declaration and the qualified type that declares the method, written $\text{M OK} \dashv \Phi.\text{C}$. There are four different rules for methods (from top to bottom in Figure 9)

1. that do not override another method and that are declared by classes,
2. that override another method and that are declared by classes,
3. that do not override another method and that are declared by refinements,
4. that override another method and that are declared by refinements.

All four rules check whether the type $\text{E}_0$ of the method body is a subtype of the declared return type $\text{B}_0$ of the method declaration. For methods that are being introduced, it is checked whether no method with an identical name has been introduced in a superclass (Rule 1) or in a predecessor in the refinement chain (Rule 3). For methods that override other methods, it is checked whether a method with identical name and signature exists in the superclass (Rule 2) or in a predecessor in the refinement chain (Rule 4).

### 2.8.3 Well-Typed FFJ Programs.

An FFJ program, consisting of a term, a class table, and a refinement table, is well-typed if

– the term is well-typed (checked using FFJ's term typing rules),
– all classes and refinements stored in the class table are well-formed (checked using FFJ's well-formedness rules), and

– the class table satisfies its sanity conditions.[4]

*2.8.4 Type Soundness of FFJ.*

The type system of FFJ is sound. We can prove this using the standard theorems of preservation and progress [71]:

THEOREM 1 (*Preservation*) If $\Gamma \vdash t : C$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : C'$ for some $C' <: C$.

THEOREM 2 (*Progress*) Suppose $t$ is a well-typed term.

1. If $t$ includes new $C_0(\bar{t}).f_i$ as a subterm, then $fields(last(C_0)) = \overline{C}\ \bar{f}$ for some $\overline{C}$ and $\bar{f}$.
2. If $t$ includes new $C_0(\bar{t}).m(\overline{u})$ as a subterm, then $mbody(m, last(C_0)) = (\overline{x}, t_0)$ and $|\overline{x}| = |\overline{u}|$ for some $\overline{x}$ and $t_0$.

We provide the proofs of the two theorems in Appendix A.

## 2.9 Differences to the Earlier Version

As stated previously, the FFJ version presented here is based on an earlier version [9], which is more verbose. The changes are summarized as follows:

– As stated previously, we removed the constructors to simplify the calculus.
– We introduced a refinement table and adapted the corresponding navigation functions. In the earlier version, these functions have been defined only semiformally.
– We simplified the field and method lookup algorithm and condensed the corresponding lookup functions.
– We revised several auxiliary predicates and added some new predicates to simplify the well-formedness rules.

## 3 Feature-Oriented Product Lines in FFJ$_{PL}$

In this section, our goal is to define a type system for feature-oriented product lines – a type system that checks whether all valid combinations of features yield well-typed programs. In this scenario, the features in question may be optional or mutually exclusive such that different combinations are possible that form different feature-oriented programs. Since there may be plenty of valid combinations, type checking all of them individually is usually not feasible.

To provide a type system for feature-oriented product lines, we need information about which combinations of features are valid (i.e., which features are mandatory, optional, and mutually exclusive), and we need to adapt the subtype and type rules of FFJ to check that there are no combinations/variants that lead to ill-typed terms. The type system guarantees that every program derived from a well-typed product line is a well-typed FFJ program. FFJ together with the type system for checking feature-oriented product lines is henceforth called FFJ$_{PL}$.

---

[4] As stated previously, a further requirement is that type rule T-SCAST does not occur in the type derivation of the program [58].

### 3.1 An Overview of Feature-Oriented Product Lines

A feature-oriented product line is made up of a set of feature modules and a feature model. The feature modules contains the features' implementation and the feature model describes how the feature modules can be combined. In contrast to the feature-oriented programs of Section 2, typically, some features are optional and some are mutually exclusive. (Also other relations such as disjunction, negation, and implication are possible [14]; we decompose them into mandatory, optional, and mutually exclusive features, as we will explain.) Generally, in a *derivation step*, a user selects a valid subset of features from which, subsequently, a feature-oriented program is derived. In our case, derivation means assembling the corresponding feature modules for a given set of features selected by the user. In Figure 10, we illustrate the process of *program derivation*.
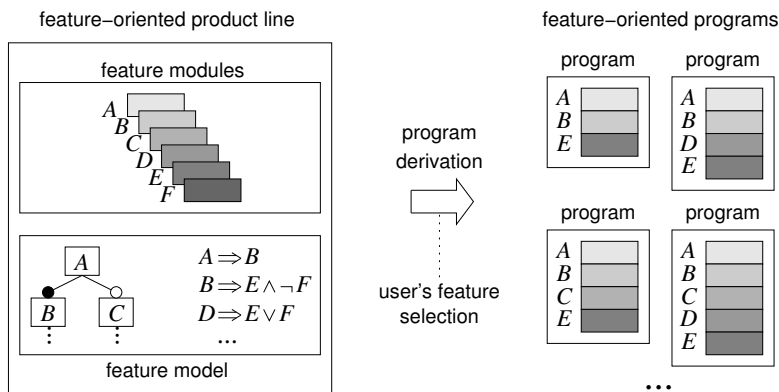


**Fig. 10** The process of deriving programs from a product line.

Typically, a wide variety of programs can be derived from a product line [23, 21]. The challenge is to define a type system that guarantees, on the basis of the feature modules and the feature model, that every valid feature combination produces a well-typed program. Once a program is derived from such a well-typed product line, we can be sure that it is well-typed, and we can evaluate it using the standard evaluation rules of FFJ (see Section 2.7).

### 3.2 Managing Variability – Feature Models

The aim of developing a product line is to manage the *variability* of a set of programs developed for a particular domain and to facilitate the *reuse* of feature implementations among the programs of the domain. A *feature model* captures the variability by (explicitly or implicitly) defining an ordered set of all features of a product line and their legal feature combinations. A well-defined feature order is essential for field and method lookup (see Section 3.7).

Different approaches to product line engineering use different representations of feature models to define legal feature combinations. The simplest approach is to enumerate all legal feature combinations. In practice, commonly different flavors of tree structures are used, sometimes in combination with additional propositional constraints, to define legal combinations [23, 14], as illustrated in Figure 10.

For our purpose, the actual representation of legal feature combinations is not relevant. In FFJ$_{PL}$, we use the feature model only to check whether features and/or specific program elements are present in certain circumstances. A design decision of FFJ$_{PL}$ is to abstract from the concrete representation of the underlying feature model and to provide an interface to the feature model instead. This has two benefits: (1) we need not to struggle with all the details of the formalization of feature models, which is well understood by researchers [14, 24, 69, 25] and outside the scope of this paper, and (2) we are able to support different kinds of feature model representations (e.g., a tree structures, grammars, or propositional formulas) [14]. The interface to the feature model is simply a set of functions and predicates that we use to ask questions like "may feature A be present together with feature B" or "is program element m present in every variant in which also feature A is present", that is, "is program element m always *reachable* from feature A".

### 3.3 Challenges of Type Checking

Let us illustrate the challenges of type checking by extending our email example, as shown in Figure 11. Suppose our basic email client is refined to process incoming text messages (feature TEXT, Lines 1–8). Optionally, it is enabled to process HTML messages, using either Mozilla's rendering engine (feature MOZILLA, Lines 9–12) or Safari's rendering engine (feature SAFARI, Lines 13–16). To this end, the features MOZILLA and SAFARI override the method render of class Display (Line 11 and 15) to invoke the respective rendering engines (field renderer, Lines 10 and 14) instead of the text printing function (Line 7).

Feature TEXT

```
1  refines class Trans {
2    Unit receive(Msg msg) {
3      return /* do something... */ new Display().render(msg);
4    }
5  }
6  class Display {
7    Unit render(Msg msg) { /* display message in text format */ }
8  }
```

Feature MOZILLA

```
9   refines class Display {
10    MozillaRenderer renderer;
11    overrides Unit render(Msg m) { /* render HTML message using the Mozilla engine */ }
12  }
```

Feature SAFARI

```
13  refines class Display {
14    SafariRenderer renderer;
15    overrides Unit render(Msg m) { /* render HTML message using the Safari engine */ }
16  }
```

**Fig. 11** A feature-oriented email client using Mozilla's and Safari's rendering engines.

The first thing to observe is that the features MOZILLA and SAFARI rely on class Display and its method render introduced by feature TEXT. To guarantee that every derived program is well-typed, the type system checks whether Display and render are *always reachable* from the features MOZILLA and SAFARI, i.e., whether, in every program variant that contains MOZILLA and SAFARI, also feature TEXT is present.

The second thing to observe is that the features MOZILLA and SAFARI both add a field renderer to Display (Lines 10 and 14), both of which have different types. In FFJ, a program with both feature modules would not be a well-typed program because the field renderer is introduced twice. However, Figure 11 is not intended to represent a single feature-oriented program but a feature-oriented product line; the features MOZILLA and SAFARI are mutually exclusive, as defined in the product line's feature model (stated earlier), and the type system has to take this fact into account.

Let us summarize the key challenges of type checking product lines:

- A global class table contains classes and refinements of all features of a product line, even if some features are optional or mutually exclusive, such that they are present only in *some* derived programs. That is, a single class can be introduced by multiple features as long as the features are mutually exclusive. This is also the case for multiple introductions of methods and fields.
- Alternative definitions of classes may have different superclasses, and alternative definitions of fields, methods, and types may have different types. Aversano et al. showed that this situation indeed occurs in practice [13] (see Section 5).
- The presence of types, fields, and methods depends on the presence of the features that introduce them. A reference from the elements of a feature to a type, field, or method is valid if the referenced element is always reachable from the referring feature. That is, the referenced element is present in every variant that contains a referring element.
- Like references, an extension of a program element, such as a class or method refinement, is valid only if the extended program element is always reachable from the feature that applies the refinement.
- Refinements of classes and methods do not necessarily form linear refinement chains. There may be alternative refinements of a single class or method that exclude one another, as we explain below.

### 3.4 Collecting Information on Feature Modules

For type checking, the $\text{FFJ}_{PL}$ compiler collects various information on the feature modules of the product line. Before the actual type checking is performed, the compiler fills three tables with information: the class table $CT$, the introduction table $IT$, and the refinement table $RT$.

The class table $CT$ of $\text{FFJ}_{PL}$ is like the one of FFJ and has to satisfy the same sanity conditions except that there may be cycles in the inheritance hierarchy, but no cycles for each set of classes which are reachable from any given feature.

The introduction table $IT$ maps a type to a list $\overline{\Phi}$ of (mutually exclusive) features that introduce the type: $IT(\mathsf{C}) = \overline{\Phi}$ for every type $\Psi.\mathsf{C} \in dom(CT)$, with $\overline{\Phi}$ being the features that introduce class $\mathsf{C}$. The features are listed in the order prescribed by the feature model. In our example of Figure 11, a call of $IT(\mathsf{Display})$ would return a list consisting only of the single feature TEXT. Likewise, the introduction table maps field and method names, in combination with their declaring classes (i.e., $IT(\mathsf{C}.\mathsf{f})$ and $IT(\mathsf{C}.\mathsf{m})$), to features. For example, a call of $IT(\mathsf{Display.renderer})$ would return the list MOZILLA, SAFARI.

Much like in FFJ, in $\text{FFJ}_{PL}$ there is a refinement table $RT$. A call of $RT(\mathsf{C})$ yields a list of all features that either introduce *or* refine class $\mathsf{C}$ (which is different from the introduction table that returns only the features that introduce class $\mathsf{C}$). As with $IT$, the features returned by $RT$ are listed in the order prescribed by the feature model.

3.5 Feature Model Interface

As stated previously, in FFJ$_{PL}$, we abstract from the concrete representation of the feature model and define instead an interface consisting of proper functions and predicates.

We would like to know which features are *sometimes* present together, which features are *never* present together, and which features are *always* present together. To this end, we define a predicate *sometimes* and a function *always*.

Predicate $sometimes(\overline{\Omega}, \Phi)$ indicates that feature $\Phi$ is sometimes present when the features $\overline{\Omega}$ are present. That is, there are variants in which the features $\overline{\Omega}$ and feature $\Phi$ are present together and there may be variants in which they are not present together. $\overline{\Omega}$ is henceforth also called *context*.

Negating the predicate (i.e., $\neg sometimes(\overline{\Omega}, \Phi)$) indicates that feature $\Phi$ is never reachable in context $\overline{\Omega}$. That is, there is no valid program variant in which the features $\overline{\Omega}$ and feature $\Phi$ are present together.

Function $always(\overline{\Omega}, \Phi)$ is used to evaluate whether feature $\Phi$ is always present in context $\overline{\Omega}$ (either alone or within a group of alternative features). The function returns a list of features. There are three cases:

1. If feature $\Phi$ is always present in the context, function *always* returns a singleton list of feature $\Phi$: $always(\overline{\Omega}, \Phi) = \Phi$.
2. If feature $\Phi$ is not always present, but belongs to a group $\overline{\Theta}$ of features, which is the smallest group in which all features are mutually exclusive and one feature of the group is always present, *always* returns this group (including $\Phi$): $always(\overline{\Omega}, \Phi) = \overline{\Theta}$ with $\Phi \in \overline{\Theta}$.
3. If the two previous cases do not apply (i.e., feature $\Phi$ is never or sometimes present and not part of a group of mutually exclusive features), *always* returns the empty list: $always(\overline{\Omega}, \Phi) = \bullet$.

The reserved feature BASE is always present.

Predicate *sometimes* and function *always* provide all information we need to know about the features' relationships. They are used especially for field and method lookup.

3.6 Valid References

We would like to know whether a specific program element is always present when a given set of features is present. This is necessary to ensure that references to program elements are always valid (i.e., not dangling). To this end, we need two sources of information. First, we need to know all features that introduce the program element in question (determined using the introduction table) and, second, we need to know which combinations of features are legal (determined using the feature model). For the field renderer of our example, the introduction table would yield the features MOZILLA and SAFARI. From the feature model, it follows that MOZILLA and SAFARI are mutually exclusive, i.e., $\neg sometimes(\text{MOZILLA}, \text{SAFARI})$. But it can happen that none of the two features is present, which can invalidate a reference to the field. The type system needs to know about this situation.

To this end, we introduce three predicates that express that certain program elements are always reachable from a set of features, shown in Figure 12. Predicate $validref_{class}(\overline{\Omega}, \mathsf{C})$ holds if type $\mathsf{C}$ is always reachable from context $\overline{\Omega}$, predicate $validref_{field}(\overline{\Omega}, \mathsf{C.f})$ holds if field $\mathsf{f}$ of class $\mathsf{C}$ is always reachable from context $\overline{\Omega}$, and predicate $validref_{method}(\overline{\Omega}, \mathsf{C.m})$

*Valid class reference* $\boxed{validref_{class}(\overline{\Omega}, \mathsf{C})}$

$$\frac{IT(\mathsf{C}) = \overline{\Phi} \qquad always(\overline{\Omega}, \Phi_1) \vee \ldots \vee always(\overline{\Omega}, \Phi_n)}{validref_{class}(\overline{\Omega}, \mathsf{C})}$$

*Valid field reference* $\boxed{validref_{field}(\overline{\Omega}, \mathsf{C})}$

$$\frac{IT(\mathsf{C.f}) = \overline{\Phi} \qquad always(\overline{\Omega}, \Phi_1) \vee \ldots \vee always(\overline{\Omega}, \Phi_n)}{validref_{field}(\overline{\Omega}, \mathsf{C.f})}$$

*Valid method reference* $\boxed{validref_{method}(\overline{\Omega}, \mathsf{C})}$

$$\frac{IT(\mathsf{C.m}) = \overline{\Phi} \qquad always(\overline{\Omega}, \Phi_1) \vee \ldots \vee always(\overline{\Omega}, \Phi_n)}{validref_{method}(\overline{\Omega}, \mathsf{C.m})}$$

**Fig. 12** Valid class, field, and method references in FFJ$_{PL}$.

*Navigating along the refinement chain*

$$\frac{RT(\mathsf{C}) = \overline{\Psi}}{last(\mathsf{C}) = \Psi_n.\mathsf{C}} \qquad \frac{RT(\mathsf{C}) = \overline{\Psi}, \Phi, \overline{\Omega} \qquad \overline{\Psi} \neq \bullet}{pred(\Phi.\mathsf{C}) = \Psi_n.\mathsf{C}} \qquad \frac{RT(\mathsf{C}) = \Phi, \overline{\Omega}}{pred(\Phi.\mathsf{C}) = \text{Base}.\mathsf{Object}}$$

**Fig. 13** Refinement in FFJ$_{PL}$.

holds if method $\mathsf{m}$ of class $\mathsf{C}$ is always reachable from context $\overline{\Omega}$. Applying *validref*[5] to a list of program elements means that the conjunction of the predicates for every list element is taken. Finally, when we write $validref_{class}(\overline{\Omega}, \mathsf{C}) \dashv \overline{\Psi}$, we mean that program element $\mathsf{C}$ is always reachable from context $\overline{\Omega}$ in a subset $\overline{\Psi}$ of features of the product line. For brevity, we do not provide a formalization here. (We need this special case in the well-formedness rules of classes and refinements.)

In our prototype, we have implemented the above functions and predicates using a SAT solver that reasons about propositional formulas representing constraints on legal feature combinations (see Section 4), as proposed by Batory [14] and Czarnecki and Pietroszek [24].

### 3.7 Refinement in FFJ$_{PL}$

In Figure 13, we show the functions *last* and *pred* for the navigation along the refinement chain. The two functions are identical to the ones of FFJ (cf. Figure 4). However, in FFJ$_{PL}$, there may be alternative declarations of a class and, in the refinement chain, refinement declarations may even precede class declarations, as long as the declaring features are mutually exclusive. Let us illustrate refinement in FFJ$_{PL}$ by means of the example shown in Figure 14. Class $\mathsf{C}$ is introduced in the features $\Phi_1$ and $\Phi_3$. Feature $\Phi_2$ refines class $\mathsf{C}$ introduced by feature $\Phi_1$ and feature $\Phi_4$ refines class $\mathsf{C}$ introduced by feature $\Phi_3$. Feature $\Phi_1$ and $\Phi_2$ are never present when feature $\Phi_3$ or $\Phi_4$ are present and vice versa. Still, a call of $RT(\mathsf{C})$ would return the list $\Phi_1, \ldots, \Phi_4$, a call of $last(\mathsf{C})$ would return the qualified type $\Phi_4.\mathsf{C}$, and a call of $pred(\Phi_4.\mathsf{C})$ would return the qualified type $\Phi_3.\mathsf{C}$ and so on.

---

[5] When we do not refer to one specific of the three predicates, we write *validref* without subscript.
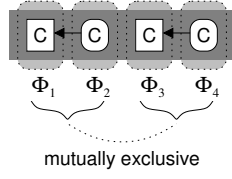
**Fig. 14** Multiple alternative refinements.

## 3.8 Subtyping in FFJ$_{PL}$

The subtype relation is more complicated in FFJ$_{PL}$ than in FFJ. The reason is that a class may have multiple declarations in different features, each declaring possibly different superclasses, as illustrated in Figure 15. That is, when checking whether a class is a subtype of another class, we need to check whether the subtype relation holds in *all* alternative inheritance paths that may be reached from a given context. For example, FooBar is a subtype of BarFoo because BarFoo is a superclass of FooBar in every program variant (since $always(\Phi_1, \Phi_2) = \Phi_2, \Phi_3$); but FooBar is not a subtype of Foo and Bar because, in both cases, a program variant exists in which FooBar is not a (indirect) subclass of the class in question.
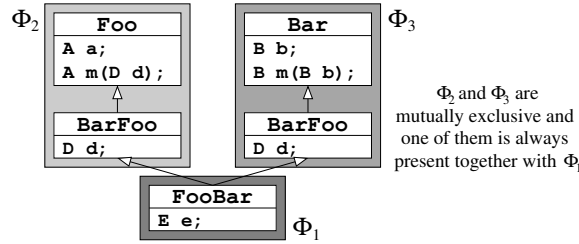


**Fig. 15** Multiple inheritance chains in the presence of alternative features.

In Figure 16, we show the subtype relation $C <: E \dashv \overline{\Omega}$ of FFJ$_{PL}$. It is read as follows: in context $\overline{\Omega}$, type C is a subtype of type E. That is, type C is a subtype of type E in every variant in which also the features of context $\overline{\Omega}$ are present. The first rule in Figure 16 covers reflexivity and terminates the recursion over the inheritance hierarchy. The second rule states that class C is a subtype of class E if at least one declaration of C is always present (tested with $validref_{class}$) and if every of C's declarations that may be present together with $\overline{\Omega}$ (tested with $sometimes$) declares some type D as its supertype and D is a subtype of E in context $\overline{\Omega}$. That is, E must be a direct or indirect supertype of D in *all* variants in which the features of context $\overline{\Omega}$ are present. Additionally, supertype D must be always reachable from context $(\overline{\Omega}, \Psi)$. Traversing the inheritance hierarchy, in each step, the context is extended by the feature that introduces the current class in question (e.g., $\overline{\Omega}$ is extended with $\Psi$).

Interestingly, the second rule subsumes the two FFJ rules for transitivity and direct superclass declaration because some declarations of C may declare E directly as its superclass, and some declarations may declare another superclass D that is, in turn, a subtype of E. The rule must be applicable to both cases simultaneously.

*Subtyping*
$$\boxed{\textsf{C} <: \textsf{E} \dashv \overline{\Omega}}$$

$$\textsf{C} <: \textsf{C} \dashv \overline{\Omega}$$

$$\frac{validref_{class}(\overline{\Omega}, \textsf{C}) \qquad \forall\, \Psi \in IT(\textsf{C})\,:\, sometimes(\overline{\Omega}, \Psi) \Rightarrow \left( \begin{array}{c} CT(\Psi.\textsf{C}) = \textsf{class C extends D \{\ldots\}} \\ validref_{class}((\overline{\Omega}, \Psi), \textsf{D}) \quad \textsf{D} <: \textsf{E} \dashv \overline{\Omega}, \Psi \end{array} \right)}{\textsf{C} <: \textsf{E} \dashv \overline{\Omega}}$$

**Fig. 16** Subtyping in FFJ$_{PL}$.

Applied to our example of Figure 15, we have $\textsf{FooBar} <: \textsf{FooBar} \dashv \Phi_1$ because of the reflexivity rule. We also have $\textsf{FooBar} <: \textsf{BarFoo} \dashv \Phi_1$ because $\textsf{FooBar}$ is reachable from feature $\Phi_1$ and every feature that introduces $\textsf{FooBar}$, namely $\Phi_1$, contains a corresponding class declaration that declares $\textsf{BarFoo}$ as $\textsf{FooBar}$'s superclass, and $\textsf{BarFoo}$ is always reachable from $\Phi_1$. However, we have $\textsf{FooBar} \not<: \textsf{Foo} \dashv \Phi_1$ and $\textsf{FooBar} \not<: \textsf{Bar} \dashv \Phi_1$ because $\textsf{FooBar}$'s immediate superclass $\textsf{BarFoo}$ is not always a subtype of $\textsf{Foo}$ respectively of $\textsf{Bar}$.

### 3.9 Auxiliary Definitions of FFJ$_{PL}$

Extending FFJ toward FFJ$_{PL}$ requires the addition and modification of some auxiliary functions. The most complex changes concern the field and method lookup mechanisms.

#### 3.9.1 Field Lookup.

The auxiliary function *fields* collects the fields of a class including the fields of its superclasses and refinements. Since alternative class or refinement declarations may introduce alternative fields (or the same field with identical or alternative types), *fields* may return different fields for different contexts. Since we want to type check all valid variants, *field* returns multiple field lists (i.e., a list of lists) that cover all possible feature selections. Each inner list contains field declarations collected in an alternative path of the combined inheritance and refinement hierarchy.

For legibility, we separate the inner lists using the delimiter '$\circ$'. For example, looking up the fields of class $\textsf{FooBar}$ in the context of feature $\Phi_1$ (Figure 15) yields the list $\textsf{A a}, \textsf{D d}, \textsf{E e} \circ \textsf{B b}, \textsf{D d}, \textsf{E e}$ because the features $\Phi_2$ and $\Phi_3$ are mutually exclusive and one of them is present in each variant in which also $\Phi_1$ is present. For readability, we use the metavariables $\mathcal{F}$ and $\mathcal{G}$ when referring to inner field lists. We abbreviate a list of lists $\mathcal{F}_1 \circ \ldots \circ \mathcal{F}_n$ of fields by $\overline{\mathcal{F}}$.

Function *fields* receives a qualified type $\Phi.\textsf{C}$ and a context $\overline{\Omega}$ of features. If we want all possible field lists, the context is empty. If we want only field lists for a subset of feature selections (e.g., only the fields that can be referenced from a term in a specific feature module), we can use the context to specify one or more features of which we know that they must be selected.

The basic idea of FFJ$_{PL}$'s field lookup is to traverse the combined inheritance and refinement hierarchy much like in FFJ. There are five situations that are handled differently:

*Field lookup*  $\boxed{fields(\overline{\Omega}, \Phi.\mathsf{C}) = \overline{\overline{\mathsf{C}\,\mathsf{f}}}}$

$$fields(\overline{\Omega}, \Phi.\mathsf{Object}) = \bullet \qquad \text{(FL-1)}$$

$$\frac{\neg sometimes(\overline{\Omega}, \Phi)}{fields(\overline{\Omega}, \Phi.\mathsf{C}) = fields(\overline{\Omega}, pred(\Phi.\mathsf{C}))} \qquad \text{(FL-2)}$$

$$\frac{always(\overline{\Omega}, \Phi) = \Phi \quad CT(\Phi.\mathsf{C}) = \mathsf{class\ C\ extends\ D\ \{}\ \overline{\mathsf{C}\,\mathsf{f}};\ \overline{\mathsf{M}}\ \}}{fields(\overline{\Omega}, \Phi.\mathsf{C}) = append(fields(\overline{\Omega}, last(\mathsf{D})), \overline{\mathsf{C}\,\mathsf{f}})} \qquad \text{(FL-3.1)}$$

$$\frac{always(\overline{\Omega}, \Phi) = \Phi \quad CT(\Phi.\mathsf{C}) = \mathsf{refines\ class\ C\ \{}\ \overline{\mathsf{C}\,\mathsf{f}};\ \overline{\mathsf{M}}\ \}}{fields(\overline{\Omega}, \Phi.\mathsf{C}) = append(fields(\overline{\Omega}, pred(\Phi.\mathsf{C})), \overline{\mathsf{C}\,\mathsf{f}})} \qquad \text{(FL-3.2)}$$

$$\frac{sometimes(\overline{\Omega}, \Phi) \quad always(\overline{\Omega}, \Phi) = \bullet \quad CT(\Phi.\mathsf{C}) = \mathsf{class\ C\ extends\ D\ \{}\ \overline{\mathsf{C}\,\mathsf{f}};\ \overline{\mathsf{M}}\ \}}{fields(\overline{\Omega}, \Phi.\mathsf{C}) = append(fields(\overline{\Omega}, last(\mathsf{D})), \overline{\mathsf{C}\,\mathsf{f}@})} \qquad \text{(FL-4.1)}$$

$$\frac{sometimes(\overline{\Omega}, \Phi) \quad always(\overline{\Omega}, \Phi) = \bullet \quad CT(\Phi.\mathsf{C}) = \mathsf{refines\ class\ C\ \{}\ \overline{\mathsf{C}\,\mathsf{f}};\ \overline{\mathsf{M}}\ \}}{fields(\overline{\Omega}, \Phi.\mathsf{C}) = append(fields(\overline{\Omega}, pred(\Phi.\mathsf{C})), \overline{\mathsf{C}\,\mathsf{f}@})} \qquad \text{(FL-4.2)}$$

$$\frac{sometimes(\overline{\Omega}, \Phi) \quad always(\overline{\Omega}, \Phi) = \overline{\Psi}}{fields(\overline{\Omega}, \Phi.\mathsf{C}) = fields((\overline{\Omega}, \Psi_1), \Phi.\mathsf{C}) \circ \ldots \circ fields((\overline{\Omega}, \Psi_n), \Phi.\mathsf{C})} \ \text{(FL-5)}$$

**Fig. 17** Field lookup in FFJ$_{PL}$.

1. The field lookup returns the empty list when it reaches Base.Object, i.e., the recursion terminates.
2. The field lookup ignores all fields that are introduced by features that are *never* present in a given context.
3. The field lookup collects all fields that are introduced by features that are *always* present in a given context. References to these fields are always valid.
4. The field lookup collects all fields that are introduced by features that *may* be present in a given context but that are not always present. In this case, a special marker @ is added to the fields in question because we cannot guarantee that a reference to this field is safe in the given context.[6] It is up to the type system to decide, based on the marker, whether this situation may provoke an error (e.g., the type system ignores the marker when looking for duplicate fields but reports an error when type checking object creations).
5. A special situation occurs when the field lookup identifies a group of alternative features. In such a group, each feature is optional and excludes every other feature of the group and at least one feature of the group is always present in a given context. Once the field lookup identifies a group of alternative features, we split the result list, each list containing the fields of a feature of the group and the fields of the original list.

To distinguish the different cases, we use the predicates and functions defined in Section 3.5 (especially *sometimes* and *always*). The definition of function *fields*, shown in Figure 17, follows the intuition described above: Once Base.Object is reached, the recursion terminates (FL-1). When a feature is never reachable in the given context, *fields* ignores

---

[6] Note that the marker @ is generated during type checking, so we do not include it in the syntax of FFJ.

*Method type lookup*  $\boxed{mtype(\overline{\Omega}, \mathsf{m}, \Phi.\mathsf{C}) = \overline{\overline{\mathsf{B}} \rightarrow \mathsf{B}_0}}$

$$mtype(\overline{\Omega}, \mathsf{m}, \mathrm{Base}.\mathsf{Object}) = \bullet \qquad (\text{ML-1})$$

$$\frac{\begin{array}{c} CT(\Phi.\mathsf{C}) = \text{class C extends D} \{ \overline{\mathsf{C}\,\mathsf{f}};\ \overline{\mathsf{M}} \} \\ \mathsf{B}_0\ \mathsf{m}(\overline{\mathsf{B}\,\mathsf{x}}) \{ \ldots \} \in \overline{\mathsf{M}} \qquad sometimes(\overline{\Omega}, \Phi) \end{array}}{mtype(\overline{\Omega}, \mathsf{m}, \Phi.\mathsf{C}) = mtype(\overline{\Omega}, \mathsf{m}, pred(\Phi.\mathsf{C})), mtype(\overline{\Omega}, \mathsf{m}, last(\mathsf{D})), \overline{\mathsf{B}} \rightarrow \mathsf{B}_0} \quad (\text{ML-2})$$

$$\frac{\begin{array}{c} CT(\Phi.\mathsf{C}) = \text{refines class C} \{ \overline{\mathsf{C}\,\mathsf{f}};\ \overline{\mathsf{M}} \} \\ \mathsf{B}_0\ \mathsf{m}(\overline{\mathsf{B}\,\mathsf{x}}) \{ \ldots \} \in \overline{\mathsf{M}} \qquad sometimes(\overline{\Omega}, \Phi) \end{array}}{mtype(\overline{\Omega}, \mathsf{m}, \Phi.\mathsf{C}) = mtype(\overline{\Omega}, \mathsf{m}, pred(\Phi.\mathsf{C})), \overline{\mathsf{B}} \rightarrow \mathsf{B}_0} \quad (\text{ML-3})$$

$$\frac{\begin{array}{c} CT(\Phi.\mathsf{C}) = \text{class C extends D} \{ \overline{\mathsf{C}\,\mathsf{f}};\ \overline{\mathsf{M}} \} \\ (\mathsf{m}\ \text{is not defined in}\ \overline{\mathsf{M}}\ \vee\ \neg sometimes(\overline{\Omega}, \Phi)) \end{array}}{mtype(\overline{\Omega}, \mathsf{m}, \Phi.\mathsf{C}) = mtype(\overline{\Omega}, \mathsf{m}, pred(\Phi.\mathsf{C})), mtype(\overline{\Omega}, \mathsf{m}, last(\mathsf{D}))} \quad (\text{ML-4})$$

$$\frac{\begin{array}{c} CT(\Phi.\mathsf{C}) = \text{refines class C} \{ \overline{\mathsf{C}\,\mathsf{f}};\ \overline{\mathsf{M}} \} \\ (\mathsf{m}\ \text{is not defined in}\ \overline{\mathsf{M}}\ \vee\ \neg sometimes(\overline{\Omega}, \Phi)) \end{array}}{mtype(\overline{\Omega}, \mathsf{m}, \Phi.\mathsf{C}) = mtype(\overline{\Omega}, \mathsf{m}, pred(\Phi.\mathsf{C}))} \quad (\text{ML-5})$$

**Fig. 18** Method type lookup in FFJ$_{PL}$.

this feature and resumes with the previous one (FL-2). When a feature is mandatory (i.e., always present in a given context), the fields in question are added to each alternative result list, which were created in Rule FL-5 (FL-3.1 and FL-3.2).[7] When a feature is optional, the fields in question, annotated with the marker @, are appended to each alternative result list (FL-4.1 and FL-4.2). When a feature is part of an alternative group of features, we cannot immediately decide how to proceed. We split the result list in multiple lists (by means of multiple recursive invocations of *fields*), in which we add one of the alternative features to each context passed to an invocation of *fields* (FL-5).

### 3.9.2 Method Type Lookup.

Like in field lookup, in method lookup, we have to take alternative definitions of methods into account. But the lookup mechanism is simpler than in *fields* because the order of signatures found in the combined inheritance and refinement hierarchy is irrelevant for type checking. Hence, function *mtype* yields a simple list $\overline{\mathsf{B}} \rightarrow \mathsf{B}_0$ of signatures for a given method name $\mathsf{m}$. For example, calling $mtype(\Phi_1, \mathsf{m}, \Phi_1.\mathsf{C})$ in the context of Figure 15 yields the list $\mathsf{D} \rightarrow \mathsf{A}, \mathsf{B} \rightarrow \mathsf{B}$.

In Figure 18, we show the definition of function *mtype*. For Base.Object, the empty list is returned (ML-1). If a class that is sometimes reachable (using *sometimes*) introduces a method in question (ML-2), its signature is added to the result list and all possible predecessors in the refinement chain (using *pred*) and all possible subclasses are searched (using *last*). Likewise, if a refinement that is sometimes reachable (using *sometimes*) introduces a method with the name searched (ML-3), its signature is added to the result list and all possible predecessors in the refinement chain are searched (using *pred*). If a class or refinement

---

[7] Function *append* adds to each inner list of a list of field lists a given field. Its implementation is straightforward and omitted for brevity.

*Valid class introduction*     $\boxed{introduce_{class}(\overline{\Omega}, \Phi.\mathsf{C})}$

$$\frac{\nexists\,\Psi\ :\ \left(\begin{array}{c} CT(\Psi.\mathsf{C}) = \mathsf{class\ C\ extends\ D\ \{\ \overline{C\,f};\ \overline{M}\ \}} \\ \Psi \neq \Phi \qquad sometimes(\overline{\Omega}, \Psi) \end{array}\right)}{introduce_{class}(\overline{\Omega}, \Phi.\mathsf{C})}$$

*Valid field introduction*     $\boxed{introduce_{field}(\overline{\Omega}, \mathsf{f}, \Phi.\mathsf{C})}$

$$\frac{\forall\,\overline{\mathsf{E\,h}} \in \mathit{fields}(\overline{\Omega}, \Phi.\mathsf{C})\ :\ \mathsf{f} \notin \overline{\mathsf{h}}}{introduce_{field}(\overline{\Omega}, \mathsf{f}, \Phi.\mathsf{C})}$$

*Valid method introduction*     $\boxed{introduce_{method}(\overline{\Omega}, \mathsf{m}, \Phi.\mathsf{C})}$

$$\frac{mtype(\overline{\Omega}, \mathsf{m}, \Phi.\mathsf{C}) = \bullet}{introduce_{method}(\overline{\Omega}, \mathsf{m}, \Phi.\mathsf{C})}$$

*Valid class refinement*     $\boxed{refine(\overline{\Omega}, \Phi.\mathsf{C})}$

$$\frac{RT(\mathsf{C}) = \overline{\Psi}, \Phi, \overline{\Pi} \qquad validref_{class}(\overline{\Omega}, \mathsf{C}) \dashv \overline{\Psi}}{refine(\overline{\Omega}, \Phi.\mathsf{C})}$$

*Valid method overriding*     $\boxed{override(\overline{\Omega}, \mathsf{m}, \Phi.\mathsf{C}, \overline{\mathsf{C}} \rightarrow \mathsf{C}_0)}$

$$\frac{\begin{array}{c} RT(\mathsf{C}) = \overline{\Psi}, \Phi, \overline{\Pi} \qquad validref_{method}(\overline{\Omega}, \mathsf{C.m}) \dashv \overline{\Psi}, \Phi \\ \forall\,\overline{\mathsf{B}} \rightarrow \mathsf{B}_0 \in mtype(\overline{\Omega}, \mathsf{m}, \Phi.\mathsf{C})\ :\ \overline{\mathsf{C}} = \overline{\mathsf{B}} \ \wedge\ \mathsf{C}_0 = \mathsf{B}_0 \end{array}}{override(\overline{\Omega}, \mathsf{m}, \Phi.\mathsf{C}, \overline{\mathsf{C}} \rightarrow \mathsf{C}_0)}$$

**Fig. 19** Valid introduction, refinement, and overriding in FFJ$_{PL}$.

does not declare a corresponding method (ML-4 and ML-5) or the a class is never reachable, the search proceeds with the possible superclasses or predecessors.

The current definition of function $mtype$ returns possibly many duplicate signatures. A straightforward optimization would be to remove duplicates before using the result list, which we omitted for simplicity.

### 3.9.3 Valid Introduction, Refinement, and Overriding.

In Figure 19, we show predicates for checking the validity of introduction, refinement, and overriding in FFJ$_{PL}$. Predicate $introduce_{class}$ indicates whether a class with the qualified type $\Phi.\mathsf{C}$ has not been introduced by any other feature $\Psi$ that may be present in context $\overline{\Omega}$. Likewise, $introduce_{field}$ and $introduce_{method}$ hold if a field $\mathsf{f}$ or a method $\mathsf{m}$ has not been introduced by a qualified type $\Phi.\mathsf{C}$ (including possible predecessors and superclasses) that may be present in the given context $\overline{\Omega}$. To this end, the former checks whether $\mathsf{f}$ is not contained in every inner list returned by $fields$, and the latter checks whether $mtype$ yields the empty list.

For a given refinement, predicate $refine$ indicates whether a proper class, which is always reachable in the given context, has been declared previously in the refinement chain. We write $validref_{class}(\overline{\Omega}, \mathsf{C}) \dashv \overline{\Psi}$ to state that a declaration of class $\mathsf{C}$ has been introduced in the set $\overline{\Psi}$ of features, which is only a subset of the features of the product line, namely the features that precede the feature that introduces class $\mathsf{C}$. Predicate $override$ indicates whether a declaration of method $\mathsf{m}$ has been introduced (and is always reachable) in some

*Term typing*  $\boxed{\Gamma \vdash t : \overline{C} \dashv \Phi}$

$$\frac{x : C \in \Gamma}{\Gamma \vdash x : C \dashv \Phi} \qquad (\text{T-VAR}_{PL})$$

$$\frac{\Gamma \vdash t_0 : \overline{E} \dashv \Phi \qquad \forall E \in \overline{E} : \mathit{validref}_{\mathit{field}}(\Phi, E.f)}{\mathit{fields}(\Phi, \overline{\mathit{last}(E)}) = \overline{\mathcal{F}, C\, f, \mathcal{G}}}{\Gamma \vdash t_0.f : C_{11}, \ldots, C_{n1}, \ldots, C_{1m}, \ldots, C_{nm} \dashv \Phi} \qquad (\text{T-FIELD}_{PL})$$

$$\frac{\Gamma \vdash t_0 : \overline{E} \dashv \Phi \qquad \Gamma \vdash \overline{\overline{t : \overline{C}}} \dashv \Phi \qquad \forall E \in \overline{E} : \mathit{validref}_{\mathit{method}}(\Phi, E.m)}{\mathit{mtype}(\Phi, m, \overline{\mathit{last}(E)}) = \overline{\overline{D} \to B} \qquad \forall \overline{C} \in \overline{\overline{C}}, \forall \overline{D} \in \overline{\overline{D}} : \overline{C <: D} \dashv \Phi}{\Gamma \vdash t_0.m(\overline{t}) : B_{11}, \ldots, B_{n1}, \ldots, B_{1m}, \ldots, B_{nm} \dashv \Phi} \qquad (\text{T-INVK}_{PL})$$

$$\frac{\Gamma \vdash \overline{t : \overline{C}} \dashv \Phi \qquad \mathit{validref}_{\mathit{class}}(\Phi, C)}{\mathit{fields}(\Phi, \mathit{last}(C)) = \overline{\mathcal{F}} \qquad @ \notin \overline{\mathcal{F}} \qquad \forall \overline{D\, g} \in \overline{\mathcal{F}}, \forall \overline{C} \in \overline{\overline{C}} : \overline{C <: D} \dashv \Phi}{\Gamma \vdash \mathsf{new}\ C(\overline{t}) : C \dashv \Phi} \qquad (\text{T-NEW}_{PL})$$

$$\frac{\Gamma \vdash t_0 : \overline{E} \dashv \Phi \qquad \mathit{validref}_{\mathit{class}}(\Phi, C)}{\forall E \in \overline{E} : (E <: C \dashv \Phi \lor C <: E \dashv \Phi)}{\Gamma \vdash (C)t_0 : C \dashv \Phi} \qquad (\text{T-UDCAST}_{PL})$$

**Fig. 20** Term typing in FFJ$_{PL}$.

feature introduced by before the feature that refines $m$ and whether every possible declaration of $m$ in any predecessor of a $\Phi.C$ has the same signature.

### 3.10 Type Relation of FFJ$_{PL}$

The type relation of FFJ$_{PL}$ consists of type rules for terms and well-formedness rules for classes, refinements, and methods, shown in Figure 20 and Figure 21.

*3.10.1 Term Typing Rules.*

A term typing judgment in FFJ$_{PL}$ is a quadruple, consisting of a typing context $\Gamma$, a term $t$, a list of types $\overline{C}$, and a feature $\Phi$ that contains the term (see Figure 20). A term can have multiple types in a product line because there may be multiple declarations of classes, fields, and methods. The list $\overline{C}$ contains all possible types a term can have.

Rule T-VAR$_{PL}$ is standard and does not refer to the feature model. It yields a list consisting only of the type of the variable in question.

Rule T-FIELD$_{PL}$ checks whether a field access $t_0.f$ is well-typed in every possible variant in which also $\Phi$ is present. Based on the possible types $\overline{E}$ of the term $t_0$ from which the field $f$ is accessed, the rule checks whether $f$ is always reachable from $\Phi$ (using $\mathit{validref}_{\mathit{field}}$).[8] Note that this is a key mechanism of FFJ$_{PL}$'s type system. It ensures that a field, being accessed, is definitely present in every valid program variant in which the field access occurs – without generating all these variants. Furthermore, all possible fields of all possible types $\overline{E}$ are assembled in a nested list $\overline{\mathcal{F}, C\, f, \mathcal{G}}$ in which $C\, f$ denotes a declaration of

---

[8] In this case, we do not need to evaluate whether the field is marked with @; predicate $\mathit{validref}_{\mathit{field}}$ ensures that the field is always reachable.

*Method typing* $\boxed{\text{M OK} \dashv \Phi.C}$

$$\frac{\begin{array}{c} \overline{x : B}, \text{this} : C \vdash t_0 : \overline{E} \dashv \Phi \qquad \forall E \in \overline{E} : E <: B_0 \dashv \Phi \\ CT(\Phi.C) = \text{class C extends D \{ } \overline{C\,f}; \overline{M} \text{ \}} \\ \mathit{validref}_{\mathit{class}}(\Phi, \overline{B}) \qquad \mathit{introduce}_{\mathit{method}}(\Phi, m, \mathit{last}(D)) \end{array}}{B_0\ m(\overline{B\ x})\ \{\ \text{return } t_0; \}\ \text{OK} \dashv \Phi.C}$$

$$\frac{\begin{array}{c} \overline{x : B}, \text{this} : C \vdash t_0 : \overline{E} \dashv \Phi \qquad \forall E \in \overline{E} : E <: B_0 \dashv \Phi \\ CT(\Phi.C) = \text{class C extends D \{ } \overline{C\,f}; \overline{M} \text{ \}} \\ \mathit{validref}_{\mathit{class}}(\Phi, \overline{B}) \qquad \mathit{override}(\Phi, m, \mathit{last}(D), \overline{B} \to B_0) \end{array}}{\text{overrides } B_0\ m(\overline{B\ x})\ \{\ \text{return } t_0; \}\ \text{OK} \dashv \Phi.C}$$

$$\frac{\begin{array}{c} \overline{x : B}, \text{this} : C \vdash t_0 : \overline{E} \dashv \Phi \qquad \forall E \in \overline{E} : E <: B_0 \dashv \Phi \\ CT(\Phi.C) = \text{refines class C \{ } \overline{C\,f}; \overline{M} \text{ \}} \\ \mathit{validref}_{\mathit{class}}(\Phi, \overline{B}) \qquad \mathit{introduce}_{\mathit{method}}(\Phi, m, \mathit{pred}(\Phi.C)) \end{array}}{B_0\ m(\overline{B\ x})\ \{\ \text{return } t_0; \}\ \text{OK} \dashv \Phi.C}$$

$$\frac{\begin{array}{c} \overline{x : B}, \text{this} : C \vdash t_0 : \overline{E} \dashv \Phi \qquad \forall E \in \overline{E} : E <: B_0 \dashv \Phi \\ CT(\Phi.C) = \text{refines class C \{ } \overline{C\,f}; \overline{M} \text{ \}} \\ \mathit{validref}_{\mathit{class}}(\Phi, \overline{B}) \qquad \mathit{override}(\Phi, m, \mathit{pred}(\Phi.C), \overline{B} \to B_0) \end{array}}{\text{overrides } B_0\ m(\overline{B\ x})\ \{\ \text{return } t_0; \}\ \text{OK} \dashv \Phi.C}$$

*Class typing* $\boxed{\text{L OK} \dashv \Phi}$

$$\frac{\begin{array}{c} \mathit{validref}_{\mathit{class}}(\Phi, D) \qquad \mathit{validref}_{\mathit{class}}(\Phi, \overline{C}) \\ \mathit{introduce}_{\mathit{class}}(\Phi, \Phi.C) \qquad \forall f \in \overline{f} : \mathit{introduce}_{\mathit{field}}(\Phi, f, \mathit{last}(D)) \qquad \overline{M}\ \text{OK} \dashv \Phi.C \end{array}}{\text{class C extends D \{ } \overline{C\,f}; \overline{M} \text{ \} OK} \dashv \Phi}$$

*Refinement typing* $\boxed{\text{R OK} \dashv \Phi}$

$$\frac{\begin{array}{c} \mathit{validref}_{\mathit{class}}(\Phi, \overline{C}) \\ \mathit{refine}(\Phi, \Phi.C) \qquad \forall f \in \overline{f} : \mathit{introduce}_{\mathit{field}}(\Phi, f, \mathit{pred}(\Phi.C)) \qquad \overline{M}\ \text{OK} \dashv \Phi.C \end{array}}{\text{refines class C \{ } \overline{C\,f}; \overline{M} \text{ \} OK} \dashv \Phi}$$

**Fig. 21** Well-formedness rules of FFJ$_{PL}$.

field f;[9] the call of $\mathit{fields}(\Phi, \overline{\mathit{last}(E)})$ is shorthand for $\mathit{fields}(\Phi, \mathit{last}(E_1)) \ldots \mathit{fields}(\Phi, \mathit{last}(E_n))$, in which the individual result lists are concatenated. Finally, the list of all possible types $C_{11}, \ldots, C_{n1}, \ldots, C_{1m}, \ldots, C_{nm}$ of field f becomes the list of types of the overall field access. Note that the result list may contain duplicates, which could be eliminated for optimization purposes.

Rule T-INVK$_{PL}$ checks whether a method invocation $t_0.m(\overline{t})$ is well-typed in every possible variant in which also $\Phi$ is present. Based on the possible types $\overline{E}$ of the term $t_0$ on which the method m is invoked, the rule checks whether m is always reachable from $\Phi$ (using $\mathit{validref}_{\mathit{method}}$). As with field access, this check is essential. It ensures that in generated programs only methods are invoked that are also present. Furthermore, all possible signatures of m of all possible types $\overline{E}$ are assembled in the nested list $\overline{\overline{D}} \to B$, and it is checked that all possible lists $\overline{C}$ of argument types of the method invocation are subtypes of all possible lists $\overline{D}$ of parameter types of the method (this implies that the lengths of the two lists must be equal). A method invocation has multiple types assembled in a list that

---

[9] For brevity, we flatten lists in the calculus if their use is unambiguous.

contains all result types of method $\mathsf{m}$ determined by $mtype$. As with field access, duplicates should be eliminated for optimization purposes.

Rule T-NEW$_{PL}$ checks whether an object creation $\mathsf{new}\ \mathsf{C}(\bar{\mathsf{t}})$ is well-typed in every possible variant in which also $\Phi$ is present. Specifically, it checks whether there is a declaration of class $\mathsf{C}$ always reachable from $\Phi$. Furthermore, all possible field combinations of $\mathsf{C}$ are assembled in the nested list $\overline{\mathcal{F}}$, and it is checked whether all possible combinations of argument types passed to the object creation are subtypes of the types of all possible field combinations (this implies that the number of arguments types must equal the number of field types). The fields of the result list must not be annotated with the marker @ since optional fields may not be present in every variant and references may become invalid (see field lookup).[10] An object creation has only a single type $\mathsf{C}$.

Rule T-UDCAST$_{PL}$ checks whether casts are well-typed in every possible variant in which also $\Phi$ is present. This is done by checking whether the type $\mathsf{C}$ the term $\mathsf{t}_0$ is cast to is always reachable from $\Phi$ and whether this type is a subtype or supertype of all possible types $\overline{\mathsf{E}}$ the term $\mathsf{t}_0$ can have.[11] We have only a single rule T-UDCAST$_{PL}$ for up- and downcasts because the list $\overline{\mathsf{E}}$ of possible types may contain super- and subtypes of $\mathsf{C}$ simultaneously. A cast yields a list containing only a single type $\mathsf{C}$.

### 3.10.2 Well-Formedness Rules.

In Figure 21, we show the well-formedness rules of classes, refinements, and methods.

Like in FFJ, the well-formedness judgment of classes and refinements is a binary relation between a class or refinement declaration and a feature. The rule of classes checks whether the superclass and all field types are always reachable from $\Phi$ (using $validref_{class}$), whether the class declaration is unique in the scope of the enclosing feature $\Phi$ (i.e., whether no other feature, that may be present together with feature $\Phi$, introduces a class with an identical name), whether none of the fields of the class declaration have been introduced before , and whether all methods are well-formed in the context of the class' qualified type. The well-formedness rule of refinements is analogous, except that the rule checks that there is *at least one* class declaration reachable that is refined and that has been introduced before the refinement (using $refine$).

The well-formedness judgment of methods is a binary relation between a method declaration and the qualified type that declares the method. Like in FFJ, there are four different rules for methods (from top to bottom in Figure 21)

1. that do not override another method and that are declared by classes,
2. that override another method and that are declared by classes,
3. that do not override another method and that are declared by refinements,
4. that override another method and that are declared by refinements.

All four rules check whether all possible types $\overline{\mathsf{E}}$ of the method body are subtypes of the declared return type $\mathsf{B}_0$ of the method and whether the argument types $\overline{\mathsf{B}}$ are always reachable from the enclosing feature $\Phi$ (using $validref_{class}$).

For methods that are introduced, it is checked whether no method with identical name has been introduced in any possible superclass (Rule 1) or in any possible predecessor in the

---

[10] The treatment of @ is semiformal but simplifies the rule. In a more formal approach, we would have to treat each field as a triple of type, name, and marker, and we would match the marker in the rule.

[11] In FFJ$_{PL}$, we do not need a rule for stupid casts. In FFJ, rule T-SCAST was necessary to formulate the preservation theorem, which we do not need in FFJ$_{PL}$.

refinement chain (Rule 3). For methods that override other methods, it is checked whether a method with identical name and signature exists in any possible superclass (Rule 2) or in any possible predecessor in the refinement chain (Rule 4).

### 3.10.3 Well-Typed FFJ$_{PL}$ Product Lines.

An FFJ$_{PL}$ product line, consisting of a term, a class table, an introduction table, and a refinement table, is well-typed if

- the term is well-typed (checked using FFJ$_{PL}$'s term typing rules),
- all classes and refinements stored in the class table are well-formed (checked using FFJ$_{PL}$'s well-formedness rules), and
- the class table satisfies its sanity conditions.[12]

## 3.11 Soundness and Completeness of FFJ$_{PL}$

What does correctness mean in the context of a product line? The product line itself is never evaluated; rather, different programs are derived that are then evaluated. Hence, the property we are interested in is that *all* programs that can be derived from a well-typed product line are in turn well-typed. Furthermore, we would like to be sure that *all* FFJ$_{PL}$ product lines, from which only well-typed FFJ programs can be derived, are well-typed. We formulate the two properties as the two theorems *Soundness of* FFJ$_{PL}$ and *Completeness of* FFJ$_{PL}$.

### 3.11.1 Soundness

THEOREM 3 (*Soundness of* FFJ$_{PL}$) Given a well-typed FFJ$_{PL}$ product line $pl$ (including a well-typed term $t$, a well-formed class table $CT$, an introduction table $IT$, a refinement table $RT$, and a consistent feature model $FM$), every program that can be derived with a valid feature selection $fs$ is a well-typed FFJ program (cf. Figure 10).

$$\frac{pl = (t, CT, IT, RT, FM) \qquad pl \text{ is well-typed} \qquad fs \text{ is valid in } FM}{derive(pl, fs) \text{ is well-typed}}$$

Function $derive$ collects the feature modules from a product line according to a user's selection $fs$ (i.e., non-selected feature modules are removed from the derived program). After this derivation step, the class table contains only classes and refinements stemming from the selected feature modules. We define a *valid feature selection* to be a list of features whose combination does not contradict the constraints implied by the feature model.

The proof idea is to show that the type derivation tree of an FFJ$_{PL}$ product line is a superimposition of multiple *type derivation slices*. As usual, the type derivation proceeds from the root (i.e., an initial type rule that checks the term and all classes and refinements of the class table) to the leaves (type rules that do not have a premise) of the type derivation tree. Each time a term has multiple types (e.g., a method has different alternative return types, which is caused by multiple mutually exclusive method declarations), the type derivation splits into multiple branches. With *branch* we refer only to locations in which the type derivation tree is split into multiple subtrees to type check multiple mutually exclusive term definitions. Each subtree from the root of the type derivation tree along the branches toward

---

[12] Again, type rule T-SCAST must not occur in the type derivations.

a leaf is a type derivation slice. Each slice corresponds to the type derivation of a feature-oriented program.

Let us illustrate the concept of a type derivation slice with a simplified example. Suppose the application of an arbitrary type rule to a term $t$ somewhere in the type derivation. Term $t$ has multiple types $\overline{C}$ due to different alternative definitions of $t$'s subterms. For simplicity, we assume here that $t$ has only a single subterm $t_0$, like in the case of a field access ($t = t_0.f$), in which the overall term $t$ has multiple types depending on $t_0$'s and $f$'s types; the rule can be easily extended to multiple subterms by adding a predicate per subterm. The type rule ensures the well-typedness of all possible variants of $t$ on the basis of the variants of $t$'s subterm $t_0$. Furthermore, the type rule checks whether a predicate $predicate$ (e.g., $C <: D$) holds for each variant of the subterm with its possible types $\overline{E}$, written $predicate(t_0 : E_i)$. The possible types $\overline{C}$ of the overall term follow in some way from the possible types $\overline{E}$ of its subterm. Predicate $validref$ is used to check whether all referenced elements and types are present in all valid variants, including different combinations of optional features. For the general case, this can be written as follows:

$$
\frac{
\begin{array}{c}
t_0 : \overline{E} \qquad predicate(t_0 : E_1) \quad predicate(t_0 : E_2) \quad \ldots \quad predicate(t_0 : E_n) \\
validref(\ldots) \qquad validref(\ldots) \qquad \ldots
\end{array}
}{
\Gamma \vdash t : \overline{C} \dashv \Phi
} \ (\text{T-}*_{PL})
$$

The different uses of $predicate$ in the premise of an FFJ$_{PL}$ type rule correspond to the branches in the type derivation that denote alternative definitions of subterms. Hence, the premise of the FFJ$_{PL}$ type rule is the conjunction of the different premises that cover the different alternative definitions of the subterms of a term.

The proof strategy is as follows. Assuming that the FFJ$_{PL}$ type system ensures that each slice is a valid FFJ type derivation (see Lemma 5 in Appendix B.1) and that each valid feature selection corresponds to a single slice (since alternative features have been removed; see Lemma 6 in Appendix B.1), each feature-oriented program that corresponds to a valid feature selection is guaranteed to be well-typed. Note that multiple valid feature selections may correspond to the same slice because of the presence of optional features. It follows that, for every valid feature selection, we derive a well-typed FFJ program – since its type derivation is valid – whose evaluation satisfies the properties of progress and preservation (see Appendix A). In Appendix B, we describe the proof idea of Theorem 3 in more detail.

### 3.11.2 Completeness

THEOREM 4 (*Completeness of* FFJ$_{PL}$) Given an FFJ$_{PL}$ product line $pl$ (including a term $t$, a class table $CT$, an introduction table $IT$, a refinement table $RT$, and a feature model $FM$), and given that *all* valid feature selections $fs$ yield well-typed FFJ programs, according to Theorem 3, $pl$ is a well-typed product line according to the rules of FFJ$_{PL}$ (with $t$ is well-typed and $CT$, $IT$, and $RT$ are well-formed).

$$
\frac{pl = (t, CT, IT, RT, FM) \quad \forall fs : (fs \text{ is valid in } FM \Rightarrow derive(pl, fs) \text{ is well-typed})}{pl \text{ is well-typed}}
$$

The proof idea is to examine three basic cases and to generalize subsequently: (1) $pl$ has only mandatory features; (2) $pl$ has only mandatory features except a single optional feature; (3) $pl$ has only mandatory features except two mutually exclusive features. All other cases can be formulated as combinations of these three basic cases. To this end, we divide the possible relations between features into three disjoint sets: (1) a feature is reachable from

another feature in *all* variants, (2) a feature is reachable from another feature in *some*, but *not in all*, variants, (3) two features are mutually exclusive. From these three possible relations, we can prove the three basic cases in isolation and, subsequently, construct a general case that can be phrased as a combination of the three basic cases. The description of the general case and the reduction finish the proof of Theorem 4. In Appendix B, we describe the proof idea of Theorem 4 in detail.

## 4 Implementation & Discussion

### 4.1 Implementation in Haskell

We have implemented FFJ and FFJ$_{PL}$ in Haskell, including a program evaluator and a type checker for product lines.[13]. The FFJ$_{PL}$ compiler expects a set of feature modules and a feature model both of which, together, represent the product line. A feature module is represented by a directory. The files found inside a directory belong to the enclosing feature module. The FFJ$_{PL}$ compiler stores this information for type checking. Each file may contain multiple classes and class refinements. In Figure 22, we show a snapshot of our test environment, which is based on Eclipse. We use Eclipse to interpret or compile our FFJ and FFJ$_{PL}$ type system and evaluator. Specifically, the figure shows the directory structure of our email system. Beside the feature implementations, file EmailClient.features (in the right widget on the bottom) contains the feature model of the product line.
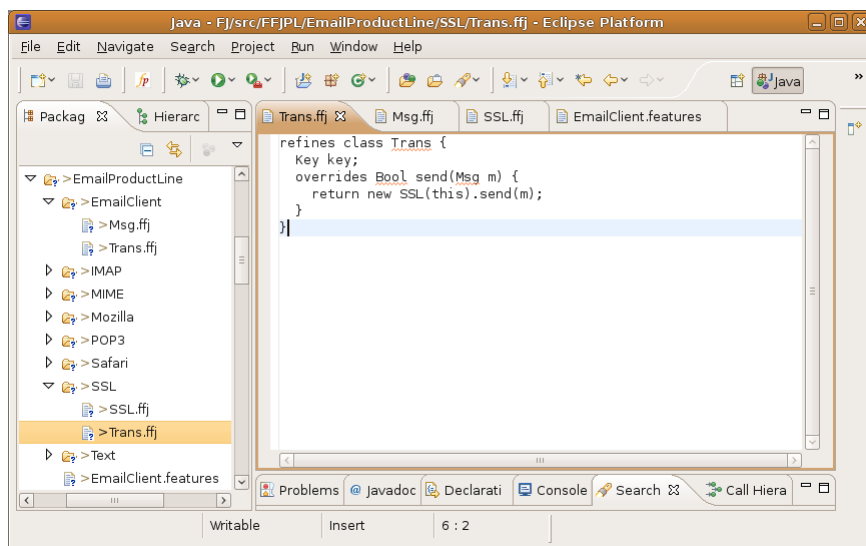


**Fig. 22** Snapshot of the test environment of the Haskell implementation (email example).

The feature model of a product line is represented by a propositional formula, following the approaches of Batory [14] and Czarnecki and Pietroszek [24]. Propositional formulas are

---

[13] http://www.fosd.de/ffj

an effective way of representing the relationships between features (e.g., of specifying which feature implies the presence and absence of other features and of machine checking whether a feature selection is valid). For example, we have implemented predicate *sometimes* as follows:[14]

$$sometimes(FM, \overline{\Omega}, \Psi) = satisfiable(FM \wedge \Omega_1 \wedge \ldots \wedge \Omega_n \wedge \Psi)$$

The feature model is a propositional formula, features are variables, and *satisfiable* is a satisfiability solver.

The implementation of function *always* is more complicated due to the case distinction (cf. Section 3.5). A simplified variant that covers only the first case (feature $\Psi$ is always present in context $\overline{\Omega}$) can be formulated as follows:

$$always_{case_1}(FM, \overline{\Omega}, \Psi) = \neg satisfiable(\neg(FM \Rightarrow ((\Omega_1 \wedge \ldots \wedge \Omega_n) \Rightarrow \Psi)))$$

For more details, we refer the interested reader to our Haskell implementation. It is based on Batory's work on the relationship between propositional formulas, feature models, and feature selections [14].

In Figure 23, we show the textual specification of the feature model of our email system, which can be passed directly to the FFJ$_{PL}$ compiler.

```
 1  features:
 2      EmailClient IMAP POP3 MIME SSL Text Mozilla Safari
 3
 4  model:
 5      EmailClient implies (IMAP or POP3);
 6      IMAP implies EmailClient;
 7      POP3 implies EmailClient;
 8      MIME implies EmailClient;
 9      SSL implies EmailClient;
10      Text implies (IMAP or POP3);
11      Mozilla implies (IMAP or POP3);
12      Safari implies (IMAP or POP3);
13      Mozilla implies (not Safari);
14      Safari implies (not Mozilla);
```

**Fig. 23** Feature model of an email client product line.

The first section ('features') of the file representing the feature model defines an ordered set of names of the features of the product line, and the second section ('model') defines constraints on the features' presence in the derived programs. In our example, each email client supports the protocols IMAP, POP3, or both. Furthermore, every feature requires the presence of the base feature EMAILCLIENT. Feature TEXT requires the presence of either IMAP or POP3 or both – the same for MOZILLA and SAFARI. Finally, feature MOZILLA requires the absence of feature SAFARI and vice versa.

On the basis of the feature modules and the feature model, FFJ$_{PL}$'s type system checks the entire product line and identifies valid program variants that still contain type errors. A SAT solver is used to check whether elements are sometimes, never, or always reachable. If an error is found, the product line is rejected as ill-formed. If not, a feature-oriented program guaranteed to be well-typed can derived on the basis of a user's feature selection.

---

[14] In the implementation, we make the dependence of the predicates and functions on the feature model *FM* explicit; for ease of presentation, the feature model was omitted as an argument in Section 3.

This program can be evaluated using the standard evaluation rules of FFJ, which we have also implemented in Haskell.

In contrast to previous work on type checking feature-oriented product lines (i.e., based on feature modules) [69, 25], our type checker provides detailed error messages. This is possible due to the fine-grained checks at the level of individual term typing and well-formedness rules. For example, if a field access succeeds only in some program variants, this fact can be reported to the user and the error message can point to the erroneous field access. Previously proposed type systems compose all code of all feature modules of a product line and extract a single propositional formula [69,25], which is checked for satisfiability. If the formula is not satisfiable (i.e., a type error has occurred), it is difficult to identify the location that has caused the error (the feature combination in question has to be compiled again). See Section 5, for a detailed discussion of related approaches.

We made several tests and experiments with our Haskell implementation. However, real-world tests did not seem sensible because of two reasons. First, like FJ, FFJ is a core language that, by its relative simplicity, is suited for the formal definition and proof of language properties – in our case, a type system and its soundness and completeness. But a core language is never suited for the development of real-world programs. This is why our examples and test programs are of similar size and complexity as the FJ examples of Pierce [58]. Second, previous work has already demonstrated that feature-oriented product lines require proper type systems and that type checking entire real-world product lines is useful [69].

Nevertheless, type checking our test programs required acceptable amounts of time (on the order of milliseconds per product line). We do not claim to be able to handle full-sized feature-oriented product lines by hand-coding them in $FFJ_{PL}$. Rather, this would require an expansion of the type system to full Java (including support for features as provided by AHEAD [15] or FeatureHouse [10]) – an enticing goal, but one for the future (especially, as Java's informal language specification [30] has 688 pages). Our work lays a foundation for implementing type systems in that it provides evidence that core feature-oriented mechanisms are type-safe.

## 4.2 Scalability

Still, we would like to make some predictions on the scalability of our approach. The novelty of our type system is that it incorporates alternative features and, consequently, alternative definitions of classes, fields, and methods. This leads to a type derivation tree with possibly multiple branches denoting alternative term types. Hence, type checking a product line with many alternative features may consume a significant amount of computation time and memory. It seems that this overhead is the price for allowing alternative implementations of program parts.

Nevertheless, our approach minimizes the overhead caused by alternative features compared to the naive approach. In the naive approach, all possible programs are derived and type checked subsequently. In our approach, we type check the entire code base of the product line and branch the type derivation only at terms that really have multiple, alternative types, and not at the level of entire program variants, as done in the naive approach. Our experience with feature-oriented product lines shows that, usually, there are not many alternative features in a product line, but mostly optional features [46, 3, 69, 40, 64, 12, 10, 6, 8, 62, 65, 61]. For example, in the Berkeley DB product line (80 000 lines of code) there are 99 feature modules, but only two pairwise alternatives [10,40]; in the Graph Product Line there are 26 feature modules, of which only three are pairwise alternatives [46, 10].

Although a previous study indicates that alternative features occur in practice that vary in type (e.g., to support different hardware platforms or libraries) [13], the most alternative features that we encountered do not vary in type. That is, there are multiple definitions of fields and methods but with equal types. For example, GPL and Berkeley DB contain alternative definitions of a few methods but only with identical signatures. Type checking these product lines with our approach, the type derivation would have almost no branches. In the naive approach, still many program variants exist due to optional features. Hence, our approach is preferable. For example, in a product line with 10 features and 100 variants, in our approach, the type system would have to check 10 feature modules (with a few branches in the type derivation leading to a few simple SAT problems to be solved; see below) and, in the naive approach, the type system would have to check 100 feature modules, because a single feature module is checked in multiple variants. For product lines with a higher degree of variability (e.g., for $2^n$ we would have 1024 variants) the benefit of our approach becomes even more significant. We believe that this benefit can make a difference in real-world product line engineering.

A further point is that almost all typing and well-formedness rules contain calls to the built-in SAT solver. This results in possibly many invocations of the SAT solver at type checking time. Determining the satisfiability of a propositional formula is in general an $\mathcal{NP}$-complete problem. However, it has been shown that the structures of propositional formulas occurring in software product lines are simple enough to scale satisfiability solving to thousands of features [51]. Furthermore, in our experiments, we have observed that many calls to the SAT solver are redundant, which is easy to see when thinking about type checking feature-oriented product lines where the presence of single types or members is checked in many type rules. Consequently, we implemented a cache that stores intermediate results to decrease the number of calls to the SAT solver to a minimum.

Finally, the implementation in Haskell helped us a lot in establishing the correctness of our type rules. It can serve other researchers to reproduce and evaluate our work and to experiment with further (feature-oriented) language mechanisms.

## 5 Related Work

We divide our discussions of related work into two parts: the implementation, formal models, and type systems of (1) feature-oriented programs and of (2) feature-oriented product lines.

### 5.1 Feature-Oriented Programs

FFJ has been inspired by several feature-oriented languages and tools, most notably by the AHEAD tool suite [15], FeatureC++ [11], FeatureHouse [10], and Prehofer's feature-oriented Java extension [59]. Their key aim 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*, which are basically directories containing software artifacts. 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 specifications in the form of JavaCC files, or build scripts and deployment descriptors in the form of XML files) [15]. 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, 10].

Another category of programming languages that provide mechanisms for the definition and extension of classes and class hierarchies includes, e.g., *ContextL* [31], *Scala* [56], and *Classbox/J* [16]. 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 non-code software artifacts cannot be included in a feature [12]. However, FFJ still models a subset of the mechanisms of these languages, in particular, class refinement.

Similarly, related work on the formalization of the key concepts underlying feature-oriented programming has not disassociated the concept of a feature from the level of code. Especially, calculi for mixins [28, 18, 1, 36], traits [45], family polymorphism and virtual classes [35, 27, 32, 20], path-dependent types [56, 55], open and dependent classes [22, 29], and nested inheritance [54] 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 compiler (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 [15, 48, 10, 43]. The calculus gDeep is most closely related to FFJ since it provides a type system for feature-oriented languages that is language-independent [5, 4]. 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 or FJ) violate the principles of feature orientation (e.g., whether methods refer to classes that have been added by other features). Due to its language independence, gDeep does not have enough information to perform such checks.

5.2 Feature-Oriented Product Lines

Our work on type checking feature-oriented product lines was motivated by the work of Thaker et al. [69]. They suggested the development of a type system for feature-oriented product lines that does not check all individual programs but the individual feature implementations. They have implemented an (incomplete) type system and, in a number of case studies on product lines, they found numerous hidden errors using their type rules. Nevertheless, the implementation of their type system is ad-hoc in the sense that it is described only informally, and they do not provide any proofs. Our type system has been inspired by their work, and we were able to provide a formalization and proofs of soundness and completeness.

In a parallel line of work, Delaware et al. have developed a formal model of a feature-oriented language, called *Lightweight Feature Java* (*LFJ*), and a type system for feature-oriented product lines [25]. Their work was also influenced by the practical work of Thaker et al. So, it is not surprising that it is closest to ours. However, there are numerous differences. First, their formal model of a feature-oriented language is based on *Lightweight Java* (*LJ*) [67] and not on *Featherweight Java* (*FJ*). While LJ is more expressive, it is also more complex. We opted for the simpler variant FJ, omitting, for example, constructors and mutable state. Second, Delaware et al. do not model feature-oriented mechanisms, such as class or method refinements, directly in the dynamic semantics of the language, which was a goal of developing FFJ. For product line type checking, they introduce a transformation step in which LFJ code is "compiled down" to LJ code (i.e., they flatten refinement chains to single classes).

Lagorio et al. have shown that a flattening semantics and a direct semantics are equivalent [44]. An advantage of a "direct" semantics is that it allows a type checking and error reporting at a finer grain. In LFJ, all feature modules are composed and a single propositional formula is generated and tested for satisfiability; if the formula is not satisfiable, it is difficult to identify precisely the point of failure. In FFJ$_{PL}$, the individual type rules consult the feature model and can point directly to the point of failure.

A further advantage of our approach is that it leaves open *when* feature composition is performed. Currently, feature composition is modeled in FFJ/FFJ$_{PL}$ as a static process done before compilation but, with our approach, it becomes possible to model dynamic feature composition at run time [63, 57] by making the class and feature tables and the feature model dynamic, i.e., by allowing them to change during a computation. With LFJ, this is not possible. Hutchins has shown that feature composition can be performed by an interpreter and partial evaluation can be used to pre-evaluate the parts of a composition that are static [33].

However, Delaware et al. have developed a machine-checked model of their type system formalized with the theorem prover Coq [17]. Our proof (sketches) are hand-written, but we have a Haskell implementation of the FFJ and FFJ$_{PL}$ calculi that we have tested thoroughly.

Even previously to the work of Thaker et al., Czarnecki et al. presented an automatic verification procedure for ensuring that no ill-formed UML (unified modeling language) model template instances will be generated from a valid feature selection [24]. That is, they type check product lines that consist not of Java programs but of UML models. They use OCL (object constraint language) constraints to express and implement a type system for UML-based product lines. In this sense, their aim is very similar to that of FFJ$_{PL}$, but limited to model artifacts (although they have proposed to generalize their work to programming languages). A further difference is that Czarnecki et al. do not organize the product line's features in feature modules but, instead, *annotate* a single superimposed model, such that the type checker (and the programmer) can infer for each model element to which feature it belongs.

Kästner et al. have developed a formal calculus CFJ and a set of type rules for an annotation-based implementation of a product line [39]. Like in the approach of Czarnecki et al., and in contrast to feature-oriented languages and tools, variability is implemented with #ifdef-like directives or similar annotations on the source code [41]. Variants are generated by conditionally removing annotated code fragments that correspond to deselected features. CFJ defines a type system for annotation-based product lines and proves that variant generation preserves typing [39, 38]. That is, all programs generated from a well-typed annotation-based product line are well-typed. For example, if a method declaration is conditionally removed, the remaining code must not reference this method. The goal of the work

on CFJ is to provide a type system for preprocessors, which are still frequently used in practice. Its focus is to create a simple and backward compatible type system that can easily be integrated into existing tool environments. CFJ does not introduce new language constructs but is designed such that annotations can be checked on top of an existing FJ or Java type system.

Recently, it has been shown that a product line based on feature modules can be transformed to a product line based on annotations [42]. That is, an $FFJ_{PL}$ product line could be transformed to a CFJ product line and then typed checked by CFJ's type system. So, why did we develop FFJ in the first place? $FFJ_{PL}$ and CFJ pursue different goals. CFJ makes several restrictions to achieve backward compatibility with Java, specifically, CFJ does not introduce any new language constructs. Backward compatibility restricts expressiveness regarding alternative features: In CFJ, even with alternative features, each term can only have a single type. In contrast, FFJ and $FFJ_{PL}$ are not backward compatible to Java anyway, since we introduce new language constructs. With $FFJ_{PL}$, we explore maximum expressiveness and allow multiple types per program element. Hence, CFJ is more restricted and cannot type check all transformed $FFJ_{PL}$ programs. Handling alternative types of a term is a major innovation of $FFJ_{PL}$ over CFJ.

Another principle problem with annotation-based product lines is that modular type checking (i.e., type checking a feature in isolation) is conceptually not possible. Although $FFJ_{PL}$ does not support modular type checking of feature modules yet, Hutchins showed that it is possible in principle [32], and in further work we intend to combine his results with $FFJ_{PL}$'s approach of product line type checking.

Finally, Aversano et al. developed a tool that collects all possible types of each variable in a given C program [13]. The background is that, using the C preprocessor, a programmer can provide different definitions of a variable and, depending on preprocessor flags, a specific definition is selected before compilation. Aversano et al. argue that knowing all possible types is essential for program comprehension and for avoiding type errors. Although developed independently, our approach is a consequent next step that guarantees type correctness in the presence of alternative variable definitions (method declarations and so on).

## 6 Conclusion

A feature-oriented product line imposes severe challenges on type checking. The naive approach of checking all individual programs of a product line is not feasible because of the combinatorial explosion of program variants. Hence, the only practical option is to check the entire code base of a product line, including all features, and, based on the information of which feature combinations are valid, to ensure that it is not possible to derive a valid program variant that contains type errors.

We have developed such a type system based on a formal model of a feature-oriented Java-like language, called Feature Featherweight Java (FFJ). A distinguishing property of our work is that we have modeled the dynamic semantics of core feature-oriented mechanisms directly, without compiling feature-oriented code down to a lower-level representation such as object-oriented Java code. The direct semantics allows us to reason about core feature-oriented mechanisms in terms of themselves rather than of generated lower-level code. A further advantage is the fine-grained error reporting and that the time of feature composition may vary between compile time and run time. Finally, our approach supports the full power of mutually exclusive features, including implications such as that terms may

have multiple type. Previous work on annotation-based product lines is limited in this respect.

Based on a valid feature selection, our type system guarantees type safety for feature-oriented product lines. That is, it ensures that every program of a well-typed feature-oriented product line is well-typed. We have shown that the type system is sound and complete. Our implementation of FFJ, including the type system for product lines ($\text{FFJ}_{PL}$), indicates the feasibility of our approach and can serve as a testbed for experimenting with further feature-oriented mechanisms.

## Acknowledgment

## References

1. D. Ancona, G. Lagorio, and E. Zucca. Jam—Designing a Java Extension with Mixins. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(5):641–712, 2003.
2. F. Anfurrutia, O. Díaz, and S. Trujillo. On Refining XML Artifacts. In *Proceedings of the International Conference on Web Engineering (ICWE)*, volume 4607 of *LNCS*, pages 473–478. Springer-Verlag, 2007.
3. S. Apel and K. Böhm. Towards the Development of Ubiquitous Middleware Product Lines. In *Software Engineering and Middleware*, volume 3437 of *LNCS*, pages 137–153. Springer-Verlag, 2004.
4. S. Apel and D. Hutchins. An Overview of the gDeep Calculus. Technical Report MIP-0712, Department of Informatics and Mathematics, University of Passau, 2007.
5. S. Apel and D. Hutchins. A Calculus for Uniform Feature Composition. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2010.
6. S. Apel, F. Janda, S. Trujillo, and C. Kästner. Model Superimposition in Software Product Lines. In *Proceedings of the International Conference on Model Transformation (ICMT)*, volume 5563 of *LNCS*, pages 4–19. Springer-Verlag, 2009.
7. S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.
8. S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Feature (De)composition in Functional Programming. In *Proceedings of the International Conference on Software Composition (SC)*, volume 5634 of *LNCS*, pages 9–26. Springer-Verlag, 2009.
9. S. Apel, C. Kästner, and C. Lengauer. 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)*, pages 101–112. ACM Press, 2008.
10. S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 221–231. IEEE CS, 2009.
11. S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, volume 3676 of *LNCS*, pages 125–140. Springer-Verlag, 2005.
12. S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Transactions on Software Engineering (TSE)*, 34(2):162–180, 2008.
13. L. Aversano, M. Di Penta, and I. Baxter. Handling Preprocessor-Conditioned Declarations. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation (SCAM)*, page 83. IEEE CS, 2002.
14. D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, volume 3714 of *LNCS*, pages 7–20. Springer-Verlag, 2005.
15. D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.
16. A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 177–189. ACM Press, 2005.

17. Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2004.

18. V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1628 of *LNCS*, pages 43–66. Springer-Verlag, 1999.

19. G. Bracha and W. Cook. Mixin-Based Inheritance. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) and International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 303–311. ACM Press, 1990.

20. D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad. Tribe: A Simple Virtual Class Calculus. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 121–134. ACM Press, 2007.

21. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

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

23. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

24. K. Czarnecki and K. Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 211–220. ACM Press, 2006.

25. B. Delaware, W. Cook, and D. Batory. Fitting the Pieces Together: A Machine-Checked Model of Safe Composition. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 243–252. ACM Press, 2009.

26. S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A Mechanism for Fine-Grained Reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, 2006.

27. E. Ernst, K. Ostermann, and W. Cook. A Virtual Class Calculus. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*, pages 270–282. ACM Press, 2006.

28. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*, pages 171–183. ACM Press, 1998.

29. V. Gasiunas, M. Mezini, and K. Ostermann. Dependent Classes. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 133–152. ACM Press, 2007.

30. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. The Java Series. Addison-Wesley, 3rd edition, 2005.

31. R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-Oriented Programming. *Journal of Object Technology (JOT)*, 7(3):125–151, 2008.

32. D. Hutchins. 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)*, pages 1–19. ACM Press, 2006.

33. D. Hutchins. *Pure Subtype Systems: A Type Theory For Extensible Software*. PhD thesis, School of Informatics, University of Edinburgh, 2009.

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

35. A. Igarashi, C. Saito, and M. Viroli. Lightweight Family Polymorphism. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *LNCS*, pages 161–177. Springer-Verlag, 2005.

36. T. Kamina and T. Tamai. McJava – A Design and Implementation of Java with Mixin-Types. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3302 of *LNCS*, pages 398–414. Springer-Verlag, 2004.

37. 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.

38. C. Kästner. *Virtual Separation of Concerns: Rehabilitating the Preprocessor*. PhD thesis, School of Computer Science, University of Magdeburg, 2010. under review.

39. C. Kästner and S. Apel. Type-Checking Software Product Lines – A Formal Approach. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 258–267. IEEE CS, 2008.

40. C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features using AspectJ. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 222–232. IEEE CS, 2007.

41. C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 311–320. ACM Press, 2008.

42. C. Kästner, S. Apel, and M. Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 157–166. ACM Press, 2009.

43. C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In *Proceedings of the International Conference on Objects, Models, Components, Patterns (TOOLS EUROPE)*, volume 33 of *LNBI*, pages 174–194. Springer-Verlag, 2009.

44. G. Lagorio, M. Servetto, and E. Zucca. Featherweight Jigsaw – A Minimal Core Calculus for Modular Composition of Classes. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 5653 of *LNCS*, pages 244–268. Springer-Verlag, 2009.

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

46. R. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *Proceedings of the International Conference on Generative and Component-Based Software Engineering (GCSE)*, volume 2186 of *LNCS*, pages 10–24. Springer-Verlag, 2001.

47. R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 169–194. Springer-Verlag, 2005.

48. R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *Proceedings of the International Symposium Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 68–77. ACM Press, 2006.

49. O. Madsen and B. Moller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 397–406. ACM Press, 1989.

50. H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *LNCS*, pages 2–28. Springer-Verlag, 2003.

51. M. Mendonca, A. Wasowski, and K. Czarnecki. SAT-based Analysis of Feature Models is Easy. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 231–240. Software Engineering Institute, Carnegie Mellon University, 2009.

52. M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 127–136. ACM Press, 2004.

53. G. Murphy, A. Lai, R. Walker, and M. Robillard. Separating Features in Source Code: An Exploratory Study. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 275–284. IEEE CS, 2001.

54. N. Nystrom, S. Chong, and A. Myers. Scalable Extensibility via Nested Inheritance. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 99–115. ACM Press, 2004.

55. M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A Nominal Theory of Objects with Dependent Types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *LNCS*, pages 201–224. Springer-Verlag, 2003.

56. M. Odersky and M. Zenger. Scalable Component Abstractions. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 41–57. ACM Press, 2005.

57. K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *LNCS*, pages 89–110. Springer-Verlag, 2002.

58. B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

59. C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 419–443. Springer-Verlag, 1997.

60. T. Reenskaug, E. Andersen, A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. Skaar, and P. Stenslet. OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. *Journal of Object-Oriented Programming (JOOP)*, 5(6):27–41, 1992.

61. M. Rosenmüller, S. Apel, T. Leich, and G. Saake. Tailor-Made Data Management for Embedded Systems: A Case Study on Berkeley DB. *Data & Knowledge Engineering (DKE)*, 68(12):1493–1512, 2009.

62. M. Rosenmüller, C. Kästner, N. Siegmund, S. Sunkle, S. Apel, T. Leich, and G. Saake. SQL á la Carte – Toward Tailor-made Data Management. In *Datenbanksysteme in Business, Technologie und Web –*

*Fachtagung des GI-Fachbereichs Datenbanken und Informationssysteme*, volume P-144 of *GI-Edition – LNI*, pages 117–136. Gesellschaft für Informatik, 2009.

63. M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. Code Generation to Support Static and Dynamic Composition of Software Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 3–12. ACM Press, 2008.

64. M. Rosenmüller, N. Siegmund, H. Schirmeier, J. Sincero, S. Apel, T. Leich, O. Spinczyk, and G. Saake. FAME-DBMS: Talor-made Data Management Solutions for Embedded Systems. In *Proceedings of the EDBT Workshop on Software Engineering for Tailor-made Data Management (SETMDM)*, pages 1–6. ACM Press, 2008.

65. N. Siegmund, C. Kästner, M. Rosenmüller, F. Heidenreich, S. Apel, and G. Saake. Bridging the Gap between Variability in Client Application and Database Schema. In *Datenbanksysteme in Business, Technologie und Web – Fachtagung des GI-Fachbereichs Datenbanken und Informationssysteme*, volume P-144 of *GI-Edition – LNI*, pages 297–306. Gesellschaft für Informatik, 2009.

66. Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

67. R. Strniša, P. Sewell, and M. Parkinson. The Java Module System: Core Design and Semantic Definition. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 499–514. ACM Press, 2007.

68. P. Tarr, H. Ossher, W. Harrison, and S. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 107–119. IEEE CS, 1999.

69. S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 95–104. ACM Press, 2007.

70. M. VanHilst and D. Notkin. Using Role Components in Implement Collaboration-based Designs. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 359–369. ACM Press, 1996.

71. A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.

## A Proof of Soundness of FFJ

Before giving the main proof, we state and proof some required lemmas.

LEMMA 1 If $mtype(\mathsf{m}, last(\mathsf{D})) = \overline{\mathsf{C}} \rightarrow \mathsf{C}_0$, then $mtype(\mathsf{m}, last(\mathsf{C})) = \overline{\mathsf{C}} \rightarrow \mathsf{C}_0$ for all $\mathsf{C} <: \mathsf{D}$.

*Proof (Lemma 1)* Straightforward induction on $mtype$. There are two cases: First, if method $\mathsf{m}$ is not defined in the declaration or in any refinement of class $\mathsf{C}$, then $mtype(\mathsf{m}, last(\mathsf{C}))$ should be the same as $mtype(\mathsf{m}, last(\mathsf{E}))$ where $CT(\Phi.\mathsf{C}) = \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{E}\ \{\ldots\}$ for some $\Phi$. This follows from the definition of $mtype$ that searches $\mathsf{E}$'s refinement chain from right to left if $\mathsf{m}$ is not declared in $\mathsf{C}$'s refinement chain. Second, if $\mathsf{m}$ is defined in the declaration or in any refinement of class $\mathsf{C}$, then $mtype(\mathsf{m}, last(\mathsf{C}))$ should also be the same as $mtype(\mathsf{m}, last(\mathsf{E}))$ with $CT(\Phi.\mathsf{C}) = \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{E}\ \{\ldots\}$ for some $\Phi$. This case is covered by the well-formedness rules for methods that use the predicate $override$ to ensure that $\mathsf{m}$ is properly overridden, i.e., the signatures of the overridden and the overriding declaration of $\mathsf{m}$ are equal, and that $\mathsf{m}$ is not introduced twice, i.e., overloading is not allowed in FFJ. $\square$

LEMMA 2 (*Term substitution preserves typing*) If $\Gamma, \overline{\mathsf{x} : \mathsf{B}} \vdash \mathsf{t} : \mathsf{D}$ and $\Gamma, \overline{\mathsf{s} : \mathsf{A}}$, where $\overline{\mathsf{A} <: \mathsf{B}}$, then $\Gamma \vdash [\overline{\mathsf{x} \mapsto \mathsf{s}}]\, \mathsf{t} : \mathsf{C}$ for some $\mathsf{C} <: \mathsf{D}$.

*Proof (Lemma 2)* By induction on the derivation of $\Gamma, \overline{\mathsf{x} : \mathsf{B}} \vdash \mathsf{t} : \mathsf{D}$.

CASE (T-VAR) $\mathsf{t} = \mathsf{x}\quad \mathsf{x} : \mathsf{D} \in \Gamma$

If $\mathsf{x} \notin \overline{\mathsf{x}}$, then the result is trivial since $[\overline{\mathsf{x} \mapsto \mathsf{s}}]\, \mathsf{x} = \mathsf{x}$.[15] On the other hand, if $\mathsf{x} = \mathsf{x}_i$ and $\mathsf{D} = \mathsf{B}_i$, then, since $[\overline{\mathsf{x} \mapsto \mathsf{s}}]\, \mathsf{x} = \mathsf{s}_i$, letting $\mathsf{C} = \mathsf{A}_i$ finishes the case.

---

[15] Note that $[\overline{\mathsf{x} \mapsto \mathsf{s}}]\, \mathsf{x}$ is an abbreviation for $[\mathsf{x}_1 \mapsto \mathsf{s}_1, \ldots, \mathsf{x}_n \mapsto \mathsf{s}_n]\, \mathsf{x}$. It means that all occurrences of the variables $\mathsf{x}_1, \ldots, \mathsf{x}_n$ in the term $\mathsf{x}$ are substituted with the corresponding terms $\mathsf{s}_1, \ldots, \mathsf{s}_n$.

CASE (T-FIELD) $t = t_0.f_i$   $\Gamma, \overline{x : B} \vdash t_0 : D_0$   $fields(last(D_0)) = \overline{C\ f}$   $D = C_i$

By the induction hypothesis, there is some $C_0$ such that $\Gamma \vdash [\overline{x \mapsto s}]\, t_0 : C_0$ and $C_0 <: D_0$. It is easy to check that $fields(last(C_0)) = fields(last(D_0)), \overline{D\ g}$ for some $\overline{D\ g}$. Therefore, by T-FIELD, $\Gamma \vdash ([\overline{x \mapsto s}]\, t_0).f_i : C_i$. The fact that the refinements of a class may add new fields does not cause problems. $\overline{D\ g}$ contains all fields that $C_0$, including all of its refinements, add to $D_0$.

CASE (T-INVK) $t = t_0.m(\overline{t})$   $\Gamma, \overline{x : B} \vdash t_0 : D_0$   $mtype(m, last(D_0)) = \overline{E} \to D$
$\Gamma, \overline{x : B} \vdash \overline{t : D}$   $\overline{D <: E}$

By the induction hypothesis, there are some $C_0$ and $\overline{C}$ such that:

$$\Gamma \vdash [\overline{x \mapsto s}]\, t_0 : C_0 \quad C_0 <: D_0 \quad \Gamma \vdash [\overline{x \mapsto s}]\, \overline{t : C} \quad \overline{C <: D}.$$

By Lemma 1, we have $mtype(m, last(C_0)) = \overline{E} \to D$. Moreover, $\overline{C <: E}$ by the transitivity of $<:$. Therefore, by T-INVK, $\Gamma \vdash [\overline{x \mapsto s}]\, t_0.m([\overline{x \mapsto s}]\, \overline{t}) : D$. The key is that subclasses and refinements may override methods but the well-formedness rules of methods ensure that the method's type is not altered, i.e., there is no overloading in FFJ.

CASE (T-NEW) $t = \text{new}\ D(\overline{t})$   $fields(last(D)) = \overline{D\ f}$   $\Gamma, \overline{x : B} \vdash \overline{t : C}$   $\overline{C <: D}$

By the induction hypothesis, $\Gamma \vdash [\overline{x \mapsto s}]\, \overline{t : E}$ for some $\overline{E}$ with $\overline{E <: C}$. We have $\overline{E <: D}$ by the transitivity of $<:$. Therefore, by rule T-NEW, $\Gamma \vdash \text{new}\ D([\overline{x \mapsto s}]\, \overline{t}) : D$. Although refinements of class $D$ may add new fields, rule T-NEW ensures that the arguments of the object creation match the overall fields of $D$, including all refinements, in number and types. That is, the number of arguments ($\overline{t}$) equals the number of fields ($\overline{f}$) which function $fields$ returns.

CASE (T-UCAST) $t = (D)t_0$   $\Gamma, \overline{x : B} \vdash t_0 : C$   $C <: D$

By the induction hypothesis, there is some $E$ such that $\Gamma \vdash [\overline{x \mapsto s}]\, t_0 : E$ and $E <: C$. We have $E <: D$ by the transitivity of $<:$, which yields $\Gamma \vdash (D)([\overline{x \mapsto s}]\, t_0) : D$ by T-UCAST.

CASE (T-DCAST) $t = (D)t_0$   $\Gamma, \overline{x : B} \vdash t_0 : C$   $D <: C$   $D \neq C$

By the induction hypothesis, there is some $E$ such that $\Gamma \vdash [\overline{x \mapsto s}]\, t_0 : E$ and $E <: C$. If $E <: D$ or $D <: E$, then $\Gamma \vdash (D)([\overline{x \mapsto s}]\, t_0) : D$ by T-UCAST or T-DCAST, respectively. If both $D \not<: E$ and $E \not<: D$, then $\Gamma \vdash (D)([\overline{x \mapsto s}]\, t_0) : D$ (with a *stupid warning*) by T-SCAST.

CASE (T-SCAST) $t = (D)t_0$   $\Gamma, \overline{x : B} \vdash t_0 : C$   $D \not<: C$   $C \not<: D$

By the induction hypothesis, there is some $E$ such that $\Gamma \vdash [\overline{x \mapsto s}]\, t_0 : E$ and $E <: C$. This means that $E \not<: D$ because, in FFJ, each class has just one superclass and, if both $E <: C$ and $E <: D$, then either $C <: D$ or $D <: C$, which contradicts the induction hypothesis. So $\Gamma \vdash (D)([\overline{x \mapsto s}]\, t_0) : D$ (with a *stupid warning*), by T-SCAST. $\square$

LEMMA 3 (*Weakening*) If $\Gamma \vdash t : C$, then $\Gamma, x : D \vdash t : C$     $x \notin dom(\Gamma)$

*Proof (Lemma 3)* Straightforward induction. The proof for FFJ is similar to the proof for FJ. $\square$

LEMMA 4 If $mtype(m, last(C_0)) = \overline{D} \to D$, and $mbody(m, last(C_0)) = (\overline{x}, t)$, then for some $D_0$ and some $C <: D$ we have $C_0 <: D_0$ and $\overline{x : D}, this : D_0 \vdash t : C$.

*Proof (Lemma 4)* By induction on the derivation of $mbody(m, last(C_0))$. The base case (in which $m$ is defined in the most specific refinement of $C_0$) is easy since $m$ is defined in $CT(last(C_0))$ and the well-formedness of the class table implies that we can derive $\overline{x : D}, this : C_0 \vdash t : C$ by the well-formedness rules of methods. The induction step is also straightforward: if $m$ is not defined in $CT(last(C_0))$, then $mbody$ searches the refinement chain from right to left; if $m$ has not been found, the superclass' refinement chain is searched. There are two subcases: first, $m$ is defined in the declaration or in any refinement of $C_0$; this case is similar to the base case. Second, $m$ is defined in a superclass $D_0$ of $C_0$ or in one of $D_0$'s refinements; in this case, the well-formedness of the class table implies that we can derive $\overline{x : D}, this : D_0 \vdash t : C$ by the well-formedness rules of methods, which finishes the case. $\square$

Note that this lemma holds because method refinements do not change the types of the arguments and the result of a method, overloading is not allowed, and this points always to the class that is introduced or refined.

THEOREM 5 (*Preservation*) If $\Gamma \vdash t : C$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : C'$ for some $C' <: C$.

*Proof (Theorem 5)* By induction on a derivation of $t \longrightarrow t'$, with a case analysis on the final rule.

CASE (E-PROJNEW) $t = $ new $C_0(\overline{v}).f_i$ $\quad t' = v_i$ $\quad fields(last(C_0)) = \overline{D\ f}$

From the shape of $t$, we see that the final rule in the derivation of $\Gamma \vdash t : C$ must be T-FIELD, with premise $\Gamma \vdash$ new $C_0(\overline{v}) : D_0$, for some $D_0$, and that $C = D_i$. Similarly, the last rule in the derivation of $\Gamma \vdash$ new $C_0(\overline{v}) : D_0$ must be T-NEW, with premises $\Gamma \vdash \overline{v : C}$ and $\overline{C <: D}$, and with $D_0 = C_0$. In particular, $\Gamma \vdash v_i : C_i$, which finishes the case, since $C_i <: D_i$.

CASE (E-INVKNEW) $t = ($new $C_0(\overline{v})).m(\overline{u})$ $\quad t' = [\overline{x \mapsto u}, $ this $\mapsto$ new $C_0(\overline{v})] t_0$
$mbody(m, last(C_0)) = (\overline{x}, t_0)$

The final rules in the derivation of $\Gamma \vdash t : C$ must be T-INVK and T-NEW, with premises $\Gamma \vdash$ new $C_0(\overline{v}) : C_0$, $\Gamma \vdash \overline{u : C}$, $\overline{C <: D}$, and $mtype(m, last(C_0)) = \overline{D} \to C$. By Lemma 4, we have $\overline{x : D}$, this $: D_0 \vdash t : B$ for some $D_0$ and $B$, with $C_0 <: D_0$ and $B <: C$. By Lemma 3, $\Gamma, \overline{x : D}$, this $: D_0 \vdash t_0 : B$. Then, by Lemma 2, we have $\Gamma [\overline{x \mapsto u}, $ this $\mapsto$ new $C_0(\overline{v})] t_0 : E$ for some $E <: B$. By the transitivity of $<:$, we obtain $E <: C$. Letting $C' = E$ completes the case.

CASE (E-CASTNEW) $t = (D)($new $C_0(\overline{v}))$ $\quad C_0 <: D$ $\quad t' = $ new $C_0(\overline{v})$

The proof of $\Gamma \vdash (D)($new $C_0(\overline{v})) : C$ must end with T-UCAST since ending with T-SCAST or T-DCAST would contradict the assumption of $C_0 <: D$. The premises of T-UCAST give us $\Gamma \vdash$ new $C_0(\overline{v}) : C_0$ and $D = C$, finishing the case.

The cases for the congruence rules are easy. We show just the case E-CAST.

CASE (E-CAST) $t = (D)t_0$ $\quad t' = (D)t_0'$ $\quad t_0 \longrightarrow t_0'$

There are three subcases according to the last typing rule used.

SUBCASE (T-UCAST) $\Gamma \vdash t_0 : C_0$ $\quad C_0 <: D$ $\quad D = C$

By the induction hypothesis, $\Gamma \vdash t_0' : C_0'$ for some $C_0' <: C_0$. By transitivity of $<:$, $C_0' <: C$. Therefore, by T-UCAST, $\Gamma \vdash (C)t_0' : C$ (with no additional *stupid warning*).

SUBCASE (T-DCAST) $\Gamma \vdash t_0 : C_0$ $\quad D <: C_0$ $\quad D = C$

By the induction hypothesis, $\Gamma \vdash t_0' : C_0'$ for some $C_0' <: C_0$. If $C_0' <: C$ or $C <: C_0'$, then $\Gamma \vdash (C)t_0' : C$ by T-UCAST or T-DCAST (without any additional *stupid warning*). On the other hand, if both $C_0' \not<: C$ or $C \not<: C_0'$, then $\Gamma \vdash (C)t_0' : C$ with a *stupid warning* by T-SCAST.

SUBCASE (T-SCAST) $\Gamma \vdash t_0 : C_0$ $\quad D \not<: C_0$ $\quad C_0 \not<: D$ $\quad D = C$

By the induction hypothesis, $\Gamma \vdash t_0' : C_0'$ for some $C_0' <: C_0$. Then, also $C_0' \not<: C$ and $C \not<: C_0'$. Therefore $\Gamma \vdash (C)t_0' : C$ with a *stupid warning*. If $C_0' \not<: C$, then $C \not<: C_0'$ since $C \not<: C_0$ and, therefore, $\Gamma \vdash (C)t_0' : C$ with *stupid warning*. If $C_0' <: C$, then $\Gamma \vdash (C)t_0' : C$ by T-UCAST (with no additional *stupid warning*). This subcase is analogous to the case T-SCAST of the proof of Lemma 2. $\square$

THEOREM 6 (*Progress*) Suppose $t$ is a well-typed term.
1. If $t$ includes new $C_0(\overline{t}).f_i$ as a subterm, then $fields(last(C_0)) = \overline{C\ f}$ for some $\overline{C}$ and $\overline{f}$.
2. If $t$ includes new $C_0(\overline{t}).m(\overline{u})$ as a subterm, then $mbody(m, last(C_0)) = (\overline{x}, t_0)$ and $|\overline{x}| = |\overline{u}|$ for some $\overline{x}$ and $t_0$.

*Proof (Theorem 6)* If $t$ has new $C_0(\overline{t}).f_i$ as a subterm, then, by well-typedness of the subterm, it is easy to check that $fields(last(C_0))$ is well-defined and $f_i$ appears in it. The fact that refinements may add fields (that have not been defined already) does not invalidate this conclusion. Note that for every field of a class, including its superclasses and all its refinements, there must be a proper argument. Similarly, if $t$ has new $C_0(\overline{t}).m(\overline{u})$ as a subterm, then it is also easy to show that $mbody(m, last(C_0)) = (\overline{x}, t_0)$ and $|\overline{x}| = |\overline{u}|$ from the fact that $mtype(m, last(C_0)) = \overline{C} \to D$ where $|\overline{x}| = |\overline{C}|$. This conclusion holds for FFJ since a method refinement must have the same signature than the method refined and overloading is not allowed. $\square$

THEOREM 7 (*Type soundness of* FFJ) If $\varnothing \vdash t : C$ and $t \longrightarrow^* t'$ with $t'$ a normal form, then $t'$ is either a value $v$ with $\varnothing \vdash v : D$ and $D <: C$, or a term containing $(D)($new $C(\overline{t}))$ in which $C \not<: D$.

*Proof (Theorem 7)* Immediate from Theorem 5 and 6. Nothing changes in the proof of Theorem 7 for FFJ compared to FJ. $\square$

## B Proofs of Soundness and Completeness of FFJ$_{PL}$

In this section, we provide proof sketches of the theorems *Soundness of* FFJ$_{PL}$ and *Completeness of* FFJ$_{PL}$. A further formalization would be desirable, but we have stopped at this point. As is often the case with formal systems, there is a trade-off between formal precision and legibility. We decided that a semi-formal development of the proof strategies are the best fit for our purposes.

### B.1 Soundness

THEOREM 8 (*Soundness of* FFJ$_{PL}$) Given a well-typed FFJ$_{PL}$ product line $pl$ (including a well-typed term t, a well-formed class table $CT$, an introduction table $IT$, a refinement table $RT$, and a consistent feature model $FM$), every program that can be derived with a valid feature selection $fs$ is a well-typed FFJ program (cf. Figure 10).

$$\frac{pl = (\mathsf{t}, CT, IT, RT, FM) \qquad pl \text{ is well-typed} \qquad fs \text{ is valid in } FM}{derive(pl, fs) \text{ is well-typed}}$$

The derived program is well-typed under the premise that the FFJ$_{PL}$ type system ensures that each slice is a valid FFJ type derivation (Lemma 5) and that each valid feature selection corresponds to a single slice (Lemma 5). Before we prove Theorem 8 we develop two required lemmas that cover the two assumptions of our proof strategy.

LEMMA 5 Given a well-typed FFJ$_{PL}$ product line, every slice of the product line's type derivation corresponds to a (set of) valid type derivation(s) in FFJ.

*Proof (Lemma 5 )* Given a well-typed FFJ$_{PL}$ product line, the corresponding type derivation consists of possibly multiple slices.

Recall that each subtree from the root of the type derivation tree along the branches induced by mutually exclusive features toward a leaf is a type derivation slice. The basic case is easy: there is only a simple derivation without branches due to mutually exclusive features (optional features may be present). In this case, each term has only a single type, which is the one that would also be determined by FFJ. Furthermore, FFJ$_{PL}$ guarantees that referenced types, methods, and fields are present in all valid variants, using predicate *validref*.

Let us illustrate this with the rule T-FIELD$_{PL}$; the other rules are analogous:

$$\frac{\Gamma \vdash \mathsf{t}_0 : \overline{\mathsf{E}} \dashv \Phi \qquad \forall \mathsf{E} \in \overline{\mathsf{E}} \; : \; validref_{field}(\Phi, \mathsf{E}.\mathsf{f})}{fields(\Phi, \overline{last(\mathsf{E})}) = \overline{\mathcal{F}, \mathsf{C} \, \mathsf{f}, \mathcal{G}}}{\Gamma \vdash \mathsf{t}_0.\mathsf{f} : \mathsf{C}_{11}, \ldots, \mathsf{C}_{n1}, \ldots, \mathsf{C}_{1m}, \ldots, \mathsf{C}_{nm} \dashv \Phi} \qquad (\text{T-FIELD}_{PL})$$

In the basic case, there are no branches in the type derivation and thus term $\mathsf{t}_0$ has only a single type $\mathsf{E}_1$. For the same reason, *fields* returns only a simple list of fields that contains the declaration of field f. Finally, T-FIELD$_{PL}$ checks whether the declaration of f is present in all valid variants (using *validref*$_{field}$). Hence, in the basic case, an FFJ$_{PL}$ derivation that ends at the rule T-FIELD$_{PL}$ is equivalent to a set of corresponding FFJ derivations that do not contain alternative and optional features and thus $\mathsf{t}_0$ has a single type, *fields* returns a simple list of fields that contains the declaration of f, and the declaration of f is present. The reason that an FFJ$_{PL}$ derivation without mutually exclusive features (i.e., a single slice) corresponds to multiple FFJ derivations is that the FFJ$_{PL}$ derivation may contain optional features whose different combinations correspond to the different FFJ derivations. Using predicate *validref*, all type rules of FFJ$_{PL}$ ensure that all possible combinations of optional features are well-typed.

In the case that there are multiple slices in the FFJ$_{PL}$ derivation, a term $\mathsf{t}_0$ may have multiple types $\overline{\mathsf{E}}$. The type rules of FFJ$_{PL}$ guarantee that every possible shape of a given term is well-typed. Each type of the term leads to a branch in the derivation tree. The premise of T-FIELD$_{PL}$ enforces that all possible shapes of a given term are well-typed by taking the conjunction of all branches of the derivation. Hence, if T-FIELD$_{PL}$ is successful, the premises of each individual branch hold, i.e., each slice corresponds to a well-typed FFJ program. Ensuring that, in the presence of optional features, all relevant subterms are well-typed (i.e., all referenced elements are present in all valid variants), a well-typed slice covers a set of well-typed FFJ derivations that correspond to different combinations of optional features, like in the basic case.

For example, in a field projection $\mathsf{t}_0.\mathsf{f}$, subterm $\mathsf{t}_0$ has multiple types $\overline{\mathsf{E}}$. For all these types, *fields* yields all possible combinations of fields declared by the variants of the types. It is checked whether, for each type

of subterm $t_0$, each combination of fields contains a proper declaration of field f. The different types of f become the possible types of the overall field projection term. Like in the basic case, it is enforced that every possible type of $t_0$ is present in all valid variants (using $validref_{class}$), so that each slice corresponds to a valid FFJ derivation, i.e., a set of derivations covering different combinations of optional features. □

LEMMA 6  Given a well-typed FFJ$_{PL}$ product line, each valid feature selection corresponds to a single slice in the corresponding type derivation.

*Proof  (Lemma 6)* By definition, a valid feature selection does not contain mutually exclusive features. Considering only a single valid feature selection, each term has only a single type. But the type derivation of the overall product line contains branches corresponding to alternative types of the terms. A successive removal of mutually exclusive features removes these branches until only a single branch remains. Consequently, a valid feature selection corresponds to a single slice. □

*Proof  (Theorem 8)* The fact that the FFJ$_{PL}$ type system ensures that each slice is a valid FFJ type derivation (Lemma 5) and that each valid feature selection corresponds to a single slice (Lemma 6), implies that each feature-oriented program that corresponds to a valid feature selection is well-typed. □

## B.2 Completeness

THEOREM 9  (*Completeness of* FFJ$_{PL}$) Given an FFJ$_{PL}$ product line $pl$ (including a term t, class, introduction, and refinement tables $CT$, $IT$, and $RT$, and a feature model $FM$), and given that *all* valid feature selections $fs$ yield well-typed FFJ programs, according to Theorem 3, $pl$ is a well-typed product line according to the rules of FFJ$_{PL}$ (with t is well-typed and $CT$, $IT$, and $RT$ are well-formed).

$$\frac{pl = (\text{t}, CT, IT, RT, FM) \quad \forall fs \: : \: (fs \text{ is valid in } FM \Rightarrow derive(pl, fs) \text{ is well-typed})}{pl \text{ is well-typed}}$$
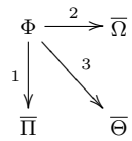
*Proof  (Theorem 9)* There are three basic cases: (1) $pl$ has only mandatory features; (2) $pl$ has only mandatory features except a single optional feature; (3) $pl$ has only mandatory features except two mutually exclusive features. Proving Theorem 9 for the first basic case is trivial. Since only mandatory features exist, only a single FFJ program can be derived from the product line. If the FFJ program is well-typed, the product line is well-typed, too, because all elements are always reachable and each term has only a single type. In fact, the type rules of FFJ$_{PL}$ and FFJ become equivalent in this case.

In the second basic case, two FFJ programs can be derived from the product line, one including and one excluding the optional feature. The difference between the two programs is the content of the optional feature. The feature can add new classes, refine existing classes by new methods and fields, and refine existing methods by overriding. If the two programs are well-typed, then the overall product line is well-typed as well since the reachability checks succeed in every type rule of FFJ$_{PL}$. Otherwise, at least one of the two programs would not be well-typed since, in this case, the reachability checks are the only difference between FFJ$_{PL}$'s and FFJ's type rules (as in the first case, each term has only a single type since there are no mutually exclusive features). The fact that the two FJ programs are well-typed implies that all elements are reachable in the type derivations of two FFJ programs. Thus, the reachability checks of the FFJ$_{PL}$ derivation succeed in every case. That is, the product line in question is well-typed.

In the third basic case, two FFJ programs can be derived from the product line, one including the first alternative and the other including the second alternative of the feature in question. The difference between the two programs is, on the one hand, the program elements one feature introduces that are not present in the other and, on the other hand, the alternative definitions of similar elements, like two alternative definitions of a single class. The first kind of difference is already covered by the second basic case. Concerning the second kind of difference: alternative definitions of a program element that are well-typed in the context of their enclosing FFJ programs, are well-typed in FFJ$_{PL}$ because they lead to two new branches in the derivation tree which are handled separately and the conjunction of their premises must hold. Since the corresponding FFJ type rule for the element succeeds in both FFJ programs, their conjunction in the FFJ$_{PL}$ type rule always holds. That is, the product line in question is well-typed.

Finally, it remains to show that all other cases (i.e., all other combinations of mandatory, optional, and alternative features) can be reduced to combinations of the three basic cases. To this end, we divide the possible relations between features into three disjoint sets: (1) a feature is reachable from another feature in *all* variants, (2) a feature is reachable from another feature in *some*, but *not in all*, variants, (3) two features are mutually exclusive. From these three possible relations we construct a general case that can be reduced to a combination of the three basic cases.

Assume a feature $\Phi$ that is mandatory with respect to a set of features $\overline{\Pi}$, that is optional with respect to a set of features $\overline{\Omega}$, and that is alternative to a set $\overline{\Theta}$ of features. We use arrows to illustrate to which of the three basic cases a pairwise relation between $\Phi$ and each element of a list is reduced:

$$
\begin{array}{ccc}
\Phi & \xrightarrow{\ 2\ } & \overline{\Omega} \\
\Big\downarrow{\scriptstyle 1} & \searrow{\scriptstyle 3} & \\
\overline{\Pi} & & \overline{\Theta}
\end{array}
$$

Such an arrow diagram can be created for every feature of a product line. The reason is that the three kinds of relations are orthogonal and there are no further relations relevant for type checking. Hence, the general case covers all possible relations between features and combinations of features. The description of the general case and the reduction finish the proof of Theorem 9. That is, FFJ$_{PL}$'s type system is complete. $\qquad\square$