Type-checking Software Product Lines – A Formal Approach

Christian Kästner

School of Computer Science
University of Magdeburg
39106 Magdeburg, Germany
kaestner@iti.cs.uni-magdeburg.de

Sven Apel

Dept. of Informatics and Mathematics
University of Passau
94030 Passau, Germany
apel@uni-passau.de

***Abstract*—A software product line (SPL) is an efficient means to generate a family of program variants for a domain from a single code base. However, because of the potentially high number of possible program variants, it is difficult to test all variants and ensure properties like type-safety for the entire SPL. While first steps to type-check an entire SPL have been taken, they are informal and incomplete. In this paper, we extend the Featherweight Java (FJ) calculus with feature annotations to be used for SPLs. By extending FJ's type system, we guarantee that – given a well-typed SPL – all possible program variants are well-typed as well. We show how results from this formalization reflect and help implementing our own language-independent SPL tool CIDE.**

I. Introduction

A *software product line (SPL)* is an efficient means to create a family of related programs for a domain [5], [28]. Instead of implementing each program from scratch, an SPL facilitates reuse by modeling a domain with features (increments in functionality relevant for stakeholders) and generating program variants from some assets that are common to the SPL [20], [5], [10].

In prior work, we developed the *Colored Integrated Development Environment (CIDE)* for safely decomposing legacy applications into features [21], [22]. CIDE is an SPL development tool, in which code fragments of the underlying code base are annotated with features. Program variants are generated by evaluating feature annotations and removing code fragments irrelevant for the variant, e.g., CIDE removes all transaction code from a database when the transaction feature is not required. The underlying principles of CIDE are similar to *'#ifdef'* statements in C's preprocessor, Frames/XVCL [19], or Gears [23], but more disciplined.

Due to the high number of variants that can be generated, SPLs are inherently complex and testing them is difficult [28], [30], [11], [7], [22]. Annotating arbitrary code fragments leads typically to errors in generated program variants. We distinguish between syntax errors (ill-formed with regard to the language's grammar), typing errors (ill-formed with regard to language semantics) and behavioral errors (ill-behaved with regard to program specification). In previous work, we addressed syntax errors intensively and prevent them in CIDE by abstracting from the textual representation of the artifacts [22]. Behavioral errors have been studied extensively in program specification and verification, although not in the context of SPLs. In this paper, we focus on typing errors.

Type errors (in statically typed languages) are typically detected during compilation after a program variant has been generated. For example, dangling method references can easily occur when annotating (and removing in some variants) a method declaration, but not the method invocations. As some program variants are never or rarely generated (e.g., only late after initial development when a new customer requires such a variant), potential typing problems might go undetected for a long time during development. Therefore, a fundamental goal is to check the entire SPL to guarantee that all possible program variants are well-typed.

First steps toward type-checking an entire SPL have already been taken [30], [11], [16], and also we already implemented some preliminary checks based on annotations in CIDE [22]. However, all previous implementations were incomplete, i.e., they covered only a few language semantics rules considered most important, but let other potential typing problems go undetected. Although, they helped to prevent several common compile-time errors, they did not *guarantee* that every variant is well-typed. Such a guarantee for Java is very difficult due to Java's complexity and its rather informal, textual specification (688 pages!) [14].

In this paper, we step back from full Java and formally extend *Featherweight Java (FJ)* – a Java subset for which type-soundness has been proved using a concise calculus [17]. Using this extended calculus named *Color Featherweight Java (CFJ)*, we can develop an SPL of which variants can be generated as FJ programs. We *formally prove* that – given a well-typed CFJ product line – all possible program variants that can be generated are well-typed FJ programs, i.e., generation preserves typing. Though this formalization covers only a small subset of Java, it provides theoretical insights about typing during the variant generation mechanism. Furthermore, we discuss how these results are helpful for developing SPL tools, especially our own tool CIDE.

II. Background

A. CIDE

CIDE was originally designed to explore how code fragments that implement a feature are scattered and interact inside legacy applications. For our analysis, we typically marked feature code on a printout with different colored text markers, one color per feature. This turned out to be very useful, so that the motivation for CIDE was to convey this color metaphor to a Java IDE based on Eclipse. In CIDE, developers assign features to code fragments, that are then highlighted with a background color in the editor (one color per feature), just as with the text marker on paper (see Fig. 1) . We refer to this process simply as *'coloring code'*. Due to this metaphor, we avoid adding explicit annotations like *'#ifdef'* to the source code, but store feature annotations separately.

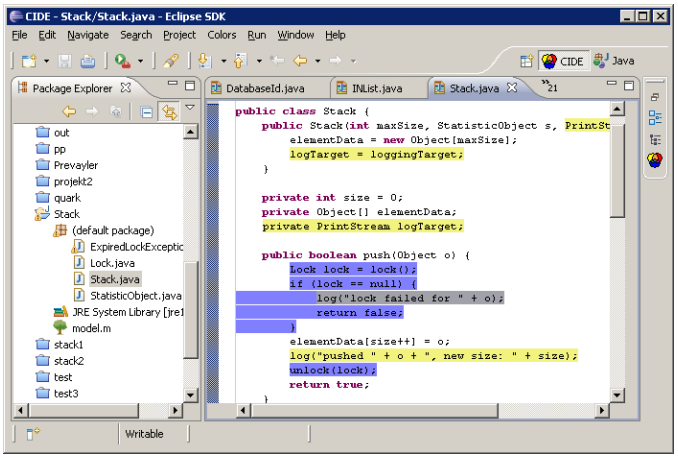

Fig. 1. CIDE Screenshot

To create a program variant, a user selects the desired features in a dialog. With this feature selection, CIDE removes all code fragments from the SPL's code base that are annotated with colors not included in this selection. For example, if a method is colored 'red' (with 'red' representing for example a transaction feature) and 'red' is not selected, then the method is removed.

Note that features may overlap, i.e., there might be code fragments that belong to multiple features. This is typical for code fragments that implement interactions (or glue code) between features [24]. For example, while in a database the features 'transactions' and 'statistics' are typically independent and optional, there might be some common code that collects statistics about transactions. This common code should only be included in the program variant if both features are included. In CIDE, the user can simply assign multiple features to a code fragment, in that case the background colors are blended.[1] When generating a variant, such glue code fragments are only included if *all* assigned features are included.

[1]Though this does not allow feature code to be recognized solely from the background colors, it indicates where a feature's code starts and ends so that the user can lookup the actual features manually via a tool tip.

CIDE's approach to annotating code from a single code base is similar to *'#ifdef'* statements in C's preprocessor, Frames/XVCL [19], or Gears [23]. Thus, it is an alternative to another category of SPL approaches that implement features in physically separated modules, e.g., using *components* [29], *aspects* [15], or *feature modules* [8], [1]. CIDE deliberately aims at *virtual separation of concerns* in that concerns (or features) are not physically modularized but just annotated, while a tool infrastructure is provided to create views on the code base.

One further concept in CIDE is important to understand our formalization. To ensure that all created variants are *syntactically correct*, CIDE only allows users to color code fragments that are optional in the language's syntax [22]. For example, in a Java file, it is possible to color entire classes, methods, statements, or parameters (when removing these elements the syntax remains valid). In contrast, it is not allowed to color arbitrary fragments, e.g., solely the name of a class (which might result in syntactically incorrect variants with source code like *'class extends Object {...}'*), or the return type of a method. Which parts are optional can be determined automatically from the language's grammar, as discussed in detail in [22].

B. Feature Model

Features and their relations in an SPL are described by a feature model [20]. The feature model describes what features are available in an SPL and for which feature combinations program variants can be created. A *feature model* consists of a set of features F and some domain constraints DC which can be described using propositional formulas [6], e.g., *'FeatureA implies FeatureB'*:

$$FM = (F, DC)$$

In practice, feature models are usually visualized by feature diagrams [20], [10], but those can be mapped to a feature set and propositional formulas as well [6].

A *program variant* is specified by a selection of features v ($v \subseteq F$). Only those variants are valid, for which the domain constraints evaluate to true ($valid(v)$).

Without domain constraints there is an exponential number of possible variants ($2^{|F|}$). With domain constraints, the number is lower, but typically still so high that generating all program variants to test them individually is not feasible [28], [30], [11], [7], [22].

C. Featherweight Java

Featherweight Java (FJ) is a minimal functional subset of the Java language that is specified formally with the FJ calculus [17], [27]. The calculus specifies FJ's type system and evaluation strategy and has been proved type-sound. It was designed to be compact; its syntax, typing rules and operational semantics fit on a single sheet of paper. FJ strips Java of many advanced features like interfaces, abstract classes, inner classes and even assignments, while retaining the core features of Java typing. There is a direct correspondence between FJ

and a purely functional core of Java, in that every FJ program is literally an executable Java program.

The motivation behind FJ was to experiment with formal extensions of Java while focusing only on the core typing features and neglecting many special cases that would require a larger calculus, without raising substantially different typing issues. Because of its simplicity even proofs for significant extensions remain manageable. For the same reasons, we chose FJ over other Java calculi like Classic Java [13], Java$_{light}$ [25], or Java^s [12]. Besides many other examples, FJ was used to formally discuss an extension of Java with generics [17], to formally discuss inner classes [18], and to reason about new composition techniques like nested inheritance [26].

In this paper, we use FJ because its compactness allows us to formally discuss typing of entire SPLs. We are not aware of any approach that discusses type safety of SPLs at a level of granularity at which even method parameters can be optional. Due to space limitations, we are not able to repeat the FJ calculus here, however its mechanisms will become clear from our formalization as we highlight our modifications and repeat unmodified rules.

III. CFJ Calculus

With CFJ, we develop a calculus for SPLs. Though it is based on FJ, it must be considered a separate language (not an extended one) to describe an entire SPL instead of a single program. SPLs written in CFJ are never directly executed, but used to generate FJ programs as, we will show later.

A. Syntax and Annotations

First, we describe CFJ's syntax and how feature annotations are introduced in this calculus. For CFJ, we use the original FJ syntax, as shown in Figure 2, with the only difference that a CFJ program consists not only of a set of classes and a term, but additionally of a feature model FM.[2] As in FJ, we require elements of lists to be named uniquely, e.g., there may not be two methods with the same name in a class. Independent of their actual storage, feature annotations are not included in the syntax of the SPL, but provided externally, as in CIDE. In our formalization, we introduce annotations using an *annotation table AT* that maps code fragments to their annotation, similar to the class table *CT* in FJ which maps class names to their declaration. Note, *AT* takes code fragments as input – which are determined by a node of the abstract syntax tree, by offset and length, or some other mechanism – instead of the name of a variable, method or class. This way, it is possible to distinguish annotations on multiple elements with the same name. The annotation table and class table are provided by the compiler.

[2]The notation in [17] and [27] differ slightly. In general, we base our description on the newer notation in [27]. Furthermore, we make slight modifications to the notation: We use $\overline{\mathsf{C}\ \mathsf{f}}$ instead of $\overline{\mathsf{C}}\ \overline{\mathsf{f}}$ to emphasize that it is a list of pairs rather than a pair of lists; the same for $\overline{\mathsf{C}\ \mathsf{x}}$ and $\overline{\mathsf{this.f=f}}$. Finally, we explicitly introduce the program P although it is technically not a syntax rule, because it would cause asymmetries in our proof otherwise.

P ::=($\overline{\mathbf{L}}$, t, FM)	*CFJ program (SPL)*
L ::=class C extends C { $\overline{\mathbf{C\ f}}$; K $\overline{\mathbf{M}}$ }	*class decl.*
K ::=C($\overline{\mathbf{C\ f}}$) { super($\bar{\mathbf{f}}$); $\overline{\mathbf{this.f=f}}$; }	*constructor decl.*
M::=C m($\overline{\mathbf{C\ x}}$) { return t; }	*method decl.*
t ::=	*terms:*
x	*variable*
t.f	*field access*
t.m($\bar{\mathbf{t}}$)	*method invocation*
new C($\bar{\mathbf{t}}$)	*object creation*
(C)t	*cast*

Fig. 2. CFJ Syntax

Following CIDE's model of guaranteeing syntactic correctness, only optional code fragments that can be removed without invalidating the syntax, can be annotated (cf. Sec. II-A). In CFJ, these are (printed bold in Fig. 2) elements of the class list ($\overline{\mathsf{L}}$), of field and parameter lists ($\overline{\mathsf{C\ f}}$ and $\overline{\mathsf{C\ x}}$), method lists ($\overline{\mathsf{M}}$), term lists ($\bar{\mathsf{t}}$), super call parameter lists ($\bar{\mathsf{f}}$), or field assignments ($\overline{\mathsf{this.f=f}}$). Because only these elements can be annotated and removed, all variants are syntactically correct although they might not be executable or well-typed, of course.

The annotation table is used the following way: $AT(\mathsf{L})$ returns the annotation of a class declaration, $AT(\mathsf{C\ f})$ returns the annotation for a field, $AT(\mathsf{C\ x})$ returns the annotation for a parameter, $AT(\mathsf{M})$ returns the annotation for a method, $AT(\mathsf{t})$ returns the annotation for a term, $AT(\mathsf{f})$ and $AT(\mathsf{this.f=f})$ return annotations for parameters and assignments inside the constructor. Furthermore, we use $AT(\mathsf{C})$ as syntactic sugar for $AT(CT(\mathsf{C}))$ to look up annotations on a class from a class name.

To be specific, annotations on terms are a bit tricky. Terms can be nested, but only few elements therein (namely parameters of method invocation and object creation) can be annotated. Therefore, in order to use $AT(\mathsf{t})$ for arbitrary terms t we may need to perform a lookup to determine the relevant annotation.[3] Consider the following example method:

```
C m() { return new C(this.a).m(new D(),new E((C) this.b)); }
```

It contains several nested terms. The method declaration contains a method invocation term of the form t.m($\bar{\mathsf{t}}$), with an object creation term (new C) as target, and two further object creation terms (new D and new E) as parameters. The first object creation term (new C) again has a field access term as parameter; the last object creation term has a cast term ((C)t) as parameter. The cast term again contains a field access term. All field access terms contain a variable this. Note, it is only possible to annotate the whole method, or to annotate method invocation parameters and object creation parameters. In our example, we underlined those parts that

[3]There are different ways of modeling annotations on terms. Another way would be to use *AT* only for elements that can be colored and define an additional auxiliary function that looks up annotations for other terms. After several rewrites we settled with the current solution to implement the lookup mechanism in the annotation table because it makes the following formalization easier.

can be annotated. To determine an annotation on a term that is not directly underlined, and thus does not have an annotation itself, the AT function now searches the abstract syntax tree upwards for the first parent that can be annotated and returns its annotation. In the given example, this.b cannot be annotated directly, thus $AT(\mathsf{this.b})$ looks up its parent: the cast term (C)t. This cast term can be annotated (it is an optional parameter of the parent constructor invocation new E(...)), thus its annotation is returned. Similarly, new C(...) cannot be annotated directly, neither can its direct parent t.m(...), but the indirect parent the method declaration C m() {...} is optional, thus $AT(\mathsf{new\ C(...)})$ returns the annotation on the method.

B. Reasoning about Annotations

So far, we did not discuss the nature of feature annotations. We are interested in expressing the following sentence *'whenever code fragment a is present, then also code fragment b is present'* based on the annotations. (We use the metavariables a and b to refer to arbitrary annotatable program fragments.) This is necessary, for example, to check whether a method invocation in code fragment a can always reference a method declaration in code fragment b, in all variants in that a is present.

There are different approaches to model annotations that allow such reasoning. As described in Section II-A, CIDE annotates a *set of features* represented by colors to each code fragment; code fragments are removed if any annotated feature is not selected in a program variant. Alternatively in [30], each code fragment is implicitly annotated with one feature; the code fragment is removed if the annotated feature is not selected and domain constraints do not say otherwise. Finally, in [11], arbitrary propositional formulas called *presence conditions* like *'FeatureA or FeatureB and not FeatureC'* are annotated; code fragments for which the annotated formula together with the domain constraints evaluates to *false* for a given feature selection are removed. Although CIDE uses a simpler mechanism, in this paper we use the most general approach of annotating propositional formulas as presence conditions. To include CIDE's simpler interpretation, we map annotated feature sets to an equivalent propositional formula in that all selected features are conjuncted (e.g., glue code annotated with $\{A,B,D\}$ as $A \wedge B \wedge D$).

Thus, $AT(\mathsf{a})$ returns a propositional formula for the code fragment a as presence condition, in which variables can be used for the features in F. This presence condition evaluates to true for exactly those program variants (specified by feature sets) in which the code fragment a should be included. We use the term *'a code fragment is present'* throughout this paper for "the code fragment's annotation evaluates to true, therefore the element is not removed in the given variant(s)".

To simplify the formalization, we incorporate the domain constraints already into AT. The propositional formula returned by AT consists of two conjuncted parts. The first part is the user-specified or generated (e.g., in CIDE) propositional formula, the second part are the SPL's domain constraints. Thus, invalid variants are not considered, i.e., they always evaluate to false for all code fragments.

The sentence *'whenever code fragment a is present, then also code fragment b is present'* can be directly expressed as implication between their propositional formulas:

$$AT(\mathsf{a}) \Rightarrow AT(\mathsf{b})$$

In the same way, the sentence *'code fragments x and y are present in the same program variants'* can be expressed as equivalence of their annotated propositional formulas:

$$(AT(\mathsf{a}) \Rightarrow AT(\mathsf{b})) \wedge (AT(\mathsf{b}) \Rightarrow AT(\mathsf{a})) \;\Leftrightarrow\; AT(\mathsf{a}) \equiv AT(\mathsf{b})$$

For comparing annotations on lists of elements, we use the short form:

$$AT(\bar{\mathsf{a}}) \Rightarrow AT(\bar{\mathsf{b}}) \quad \Leftrightarrow \\ (AT(\mathsf{a}_1) \Rightarrow AT(\mathsf{b}_1)) \;\wedge \ldots \wedge\; (AT(\mathsf{a}_n) \Rightarrow AT(\mathsf{b}_n))$$

C. Annotation Rules

Before we formally model the annotation checks as extensions to FJ's typing rules, we first informally introduce the annotation rules that are to be checked. In general, we need to check code fragments that *reference* other code fragments. The code fragments – references and targets – must be annotated in such a way that no reference is ever present in those program variants in which the target is removed. Otherwise we get dangling references that typically result in ill-typed programs. We start with this informal list of annotation rules, then model them formally in CFJ's type system, and later prove them to be complete. We identified checks for thirteen different pairs of references and targets:

- **(L.1)** A *class* L can only extend a class that is present.
- **(L.2)** A *field* C f can only have a type C of a class L that is present.
- **(K.1)** A *super constructor call (i)* can only pass those parameters that are bound to constructor parameters and *(ii)* must pass exactly the number of parameters expected in the super constructor.
- **(K.2)** A *field assignment* this.f=f in a constructor *(i)* can only access present fields C f in the same class and *(ii)* only assign values that are bound to constructor parameters.
- **(K.3)** A *constructor parameter* C f can only have a type C of a class L that is present.
- **(M.1)** A *method declaration* C m($\overline{\mathsf{C\ x}}$)... can only have a return type C of a class L that is present.
- **(M.2)** A method declaration *overriding* another method declaration must have the same signature in all variants in which both are present.
- **(M.3)** A *method declaration parameter* C x can only have a type C of a class L that is present.
- **(T.1)** A *variable* x can only be bound to parameters $\overline{\mathsf{C\ x}}$ of its enclosing method.
- **(T.2)** A *field access* t.f can only access fields $\overline{\mathsf{C\ f}}$ that are present in the enclosing class or its superclasses.
- **(T.3)** A *method invocation* t.m($\bar{\mathsf{t}}$) *(i)* can only invoke a method M that is present and *(ii)* must pass exactly the number of parameters $\overline{\mathsf{C\ x}}$ as expected by this method.

(T.4) An *object creation* $\mathsf{new\ C}(\overline{\mathsf{t}})$ *(i)* can create only objects from classes L that are present and *(ii)* must pass exactly the number of parameters $\overline{\mathsf{C\ f}}$ as expected by the target's constructor.

(T.5) A *cast* (C)t can cast a term only to a class L that is present.

Furthermore, there are some rules (which we refer to as *subtree rules* [21], [22]) that deal with the removal process of children from their parent element: if a class is removed also all methods therein must be removed, if a method is removed also its parameters and term must be removed. These rules seem obvious and are actually enforced automatically in CIDE by propagating colors from parent to child structures (e.g., if a class is 'green', then all its methods are automatically 'green' as well). However, when formalizing the calculus with propositional formulas, we have to make these rules explicit:

(SL.1) A *field* is only present when the enclosing class is present.

(SL.2) A *method* is only present when the enclosing class is present.

(SK.1) A *constructor parameter* is only present when the enclosing class is present.

(SK.2) A *super constructor invocation parameter* is only present when the enclosing class is present.

(SK.3) A *field assignment* is only present when the enclosing class is present.

(SM.1) A *method parameter* is only present when the enclosing method is present.

(ST.1) A *method invocation parameter* is only present when the enclosing method or term is present.

(ST.2) An *object creation parameter* is only present when the enclosing method or term is present.

In the remainder of this section, we highlight changes compared to the original FJ calculus for the annotation rules (L.1–T.5) in light gray and changes for the subtree rules (SL.1–ST.2) in darker gray.

D. Subtyping

The subtyping relation $<:$ in CFJ is identical to FJ. Though we could check the annotation rule (L.1) here, we decided to postpone this check to FJ's typing rules instead.

$$\mathsf{C} <: \mathsf{C} \qquad \frac{\mathsf{C} <: \mathsf{D} \quad \mathsf{D} <: \mathsf{E}}{\mathsf{C} <: \mathsf{E}} \qquad \frac{\mathsf{class\ C\ extends\ D}\ \{\ldots\}}{\mathsf{C} <: \mathsf{D}}$$

E. Auxiliary functions

As in FJ, we need some auxiliary definitions for the typing and evaluation rules. While we try to check most color checks in the typing rules, there are cases in which already the auxiliary functions – that are used in FJ to recursively lookup fields or methods across the inheritance hierarchy – need to consider annotations. We use $\mathcal{A}$ as a metavariable for annotations (i.e., presence conditions in the form of a propositional formula) and use $\bullet$ to denote an empty sequence.

1) Filter: First, we define a new function *filter*, that returns only those code fragments from a given list that are present for a given annotation $\mathcal{A}$. Specifically, only those code fragments are returned for which the annotation always evaluates to *true* when the given annotation $\mathcal{A}$ evaluates to *true*.

$$\mathit{filter}(\overline{\mathsf{a}}, \mathcal{A}) = \{\mathsf{a}_i \mid \mathcal{A} \Rightarrow AT(\mathsf{a}_i)\}$$

Note, by integrating domain constraints into the result of *AT* (see Sec. III-B), we can check the presence of a code fragment simply with $\mathcal{A} \Rightarrow AT(\mathsf{a})$, and we do not need to check them explicitly in every annotation check, e.g., $\mathcal{A} \Rightarrow AT(\mathsf{a}) \wedge DC$.

2) Field lookup: A field lookup determines all fields of a class C including fields inherited from superclasses. In CFJ this function *fields* is identical to FJ. Annotations are checked later in the typing rules.

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

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\{\overline{\mathsf{C\ f}}; \mathsf{K}\ \overline{\mathsf{M}}\} \quad \mathit{fields}(\mathsf{D}) = \overline{\mathsf{D\ g}}}{\mathit{fields}(\mathsf{C}) = \overline{\mathsf{C\ f}}, \overline{\mathsf{D\ g}}}$$

3) Method lookup: Similar to field lookups, a method lookup function *mtype* finds methods with a given name m in a class C or its superclasses. In contrast to fields, method lookups need to be adapted because of the possibility of method overriding (in contrast to overshadowing fields, which is not allowed in FJ [17]). Thus, it could be possible, that a method m in class C is not present for a given annotation $\mathcal{A}$, but another method in a superclass is present. Therefore, we cannot check annotations only in the typing rules but have to adapt the auxiliary function *mtype*. Compared to FJ, we introduce an annotation parameter and select only those methods in a class which are not filtered by the *filter* function for the given annotation. These extensions are later needed for rules (M.2) and (T.3).

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

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\{\overline{\mathsf{C\ f}}; \mathsf{K}\ \overline{\mathsf{M}}\} \quad \mathsf{m\ is\ not\ defined\ in}\ \mathit{filter}(\overline{\mathsf{M}}, \mathcal{A})}{\mathit{mtype}(\mathsf{m}, \mathsf{C}, \mathcal{A}) = \mathit{mtype}(\mathsf{m}, \mathsf{D}, \mathcal{A})}$$

4) Overriding: Finally, we extend the auxiliary function *override* that determines valid overriding for a given method m with annotation $\mathcal{A}$ and type $\overline{\mathsf{C}} \to \mathsf{C}$ and a superclass D. Introducing a method in a class is valid *(i)* if there is no method with the same name in a superclass, or *(ii)* if the method in the superclass has exactly the same signature (same return type and parameter types).

In the presence of annotations, checking valid overriding is trickier than expected. First, the check whether a method with the same name exists in a superclass must be extended to check only for those methods that are present regarding a given annotation $\mathcal{A}$. This is automatically done using the modified *mtype* function. Second, return types cannot be annotated, thus their check is identical to FJ. However, checking the parameter types is more difficult. We need to make sure that

the presence of parameters match in every program variant in which overriding occurs (M.2). Therefore, to check that parameters $\overline{\mathsf{C}}$ in the given method and parameters $\overline{\mathsf{D}}$ from the overridden method are always present at the same time, we use the expression $\mathcal{A} \Rightarrow (AT(\overline{\mathsf{C}}) \equiv AT(\overline{\mathsf{D}}))$.

$$\frac{\begin{array}{c}\mathit{mtype}(\mathsf{m},\mathsf{D},\mathcal{A}) = \overline{\mathsf{D}} \to \mathsf{C}_0 \text{ implies} \\ \overline{\mathsf{C}} = \overline{\mathsf{D}} \text{ and } \mathsf{C}_0 = \mathsf{D}_0 \text{ and } \mathcal{A} \Rightarrow (AT(\overline{\mathsf{C}}) \equiv AT(\overline{\mathsf{D}}))\end{array}}{\mathit{override}(\mathsf{m},\mathsf{D},\overline{\mathsf{C}} \to \mathsf{C}_0,\mathcal{A})}$$

F. Typing

After defining auxiliary functions, we can finally extend FJ's typing to incorporate annotations. We revisit each typing rule in FJ and adapt it if necessary. For brevity, we discuss only the changes compared to the original FJ typing rules.

1) T-VAR*:* Typing of variables is invariant to annotations, i.e., the same as in FJ.

$$\frac{\mathsf{x} : \mathsf{C} \in \Gamma}{\Gamma \vdash \mathsf{x} : \mathsf{C}}$$

2) T-FIELD*:* For typing of field accesses, we require that the target field declaration is present (T.2). Therefore, we look up the current annotation of the field access term ($AT(\mathsf{t}_0.\mathsf{f}_i) = \mathcal{A}$) and combine the *filter* and *fields* functions to retrieve only fields that are present in the target class. Furthermore, we check that the class corresponding to the field's type (C_i) is present (L.2).

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \quad AT(\mathsf{t}_0.\mathsf{f}_i) = \mathcal{A} \quad \mathit{filter}(\mathit{fields}(\mathsf{C}_0),\mathcal{A}) = \overline{\mathsf{C}\ \mathsf{f}} \\ AT(\mathsf{C}_i\ \mathsf{f}_i) \Rightarrow AT(\mathsf{C}_i)\end{array}}{\Gamma \vdash \mathsf{t}_0.\mathsf{f}_i : \mathsf{C}_i}$$

3) T-INVK*:* Method invocations are checked similarly for the target method to be present (T.3*i*) using the filtering of *mtype*. Additionally for rule (T.3*ii*), the annotation of the invocation parameters must match the annotation of the parameter declaration as described in Section III-E4 for the *override* function ($\mathcal{A} \Rightarrow (AT(\bar{\mathsf{t}}) \equiv AT(\overline{\mathsf{D}}))$). Finally, the subtree rule (ST.1) is checked, i.e., parameters are at most present when the term itself is present.

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \quad AT(\mathsf{t}_0.\mathsf{m}(\bar{\mathsf{t}})) = \mathcal{A} \quad \mathit{mtype}(\mathsf{m},\mathsf{C}_0,\mathcal{A}) = \overline{\mathsf{D}} \to \mathsf{C} \\ \Gamma \vdash \bar{\mathsf{t}} : \overline{\mathsf{C}} \quad \overline{\mathsf{C}} <: \overline{\mathsf{D}} \quad \mathcal{A} \Rightarrow (AT(\bar{\mathsf{t}}) \equiv AT(\overline{\mathsf{D}})) \quad AT(\bar{\mathsf{t}}) \Rightarrow \mathcal{A}\end{array}}{\Gamma \vdash \mathsf{t}_0.\mathsf{m}(\bar{\mathsf{t}}) : \mathsf{C}}$$

4) T-NEW*:* The extensions for typing object creation is similar to method invocations. First, the target class must be present (T.4*i*), which is checked explicitly with $\mathcal{A} \Rightarrow AT(C)$. Additionally for rule (T.4*ii*), the provided parameters must match the expected constructor parameters ($\mathcal{A} \Rightarrow (AT(\bar{\mathsf{t}}) \equiv AT(\overline{\mathsf{D}\ \mathsf{f}}))$), in which additionally *filter* is used to compare only present fields. Finally, the subtree rule (ST.2) is checked.

$$\frac{\begin{array}{c}AT(\mathsf{new}\ \mathsf{C}(\bar{\mathsf{t}})) = \mathcal{A} \quad \mathit{filter}(\mathit{fields}(\mathsf{C}),\mathcal{A}) = \overline{\mathsf{D}\ \mathsf{f}} \\ \Gamma \vdash \bar{\mathsf{t}} : \overline{\mathsf{C}} \quad \overline{\mathsf{C}} <: \overline{\mathsf{D}} \\ \mathcal{A} \Rightarrow AT(\mathsf{C}) \quad \mathcal{A} \Rightarrow (AT(\bar{\mathsf{t}}) \equiv AT(\overline{\mathsf{D}\ \mathsf{f}})) \quad AT(\bar{\mathsf{t}}) \Rightarrow \mathcal{A}\end{array}}{\Gamma \vdash \mathsf{new}\ \mathsf{C}(\bar{\mathsf{t}}) : \mathsf{C}}$$

*5) Casts (*T-UCAST*,* T-DCAST*):* For casts, we need to check that the target class is present (T.5) as shown below. Stupid casts (needed for FJ's small step semantics) can, but do not need to be extended because they result in ill-typed programs anyway, see [17]. This just leaves us with straightforward extensions of T-UCAST and T-DCAST:

$$\frac{\Gamma \vdash \mathsf{t}_0 : \mathsf{D} \quad \mathsf{D} <: \mathsf{C} \quad AT((\mathsf{C})\mathsf{t}_0) \Rightarrow AT(\mathsf{C})}{\Gamma \vdash (\mathsf{C})\mathsf{t}_0 : \mathsf{C}} \qquad \frac{\Gamma \vdash \mathsf{t}_0 : \mathsf{D} \quad \mathsf{C} <: \mathsf{D} \quad \mathsf{C} \neq \mathsf{D} \quad AT((\mathsf{C})\mathsf{t}_0) \Rightarrow AT(\mathsf{C})}{\Gamma \vdash (\mathsf{C})\mathsf{t}_0 : \mathsf{C}}$$

6) Method Typing (M OK in C): For method typing, we pass the annotation of the method to *override* to check rule (M.2). Second, we check that the class corresponding to the return type of the method (C_0) is present (M.1). Third, we check for each parameter, that the referenced class is present ($AT(\overline{\mathsf{C}\ \mathsf{x}}) \Rightarrow AT(\overline{\mathsf{C}})$) for rule (M.3). Fourth, we provide only those parameters in the type context that are present (T.1), therefore we filter the parameters $\overline{\mathsf{C}\ \mathsf{x}}$ and provide only the present parameters $\overline{\mathsf{D}\ \mathsf{y}}$ to type-check t_0. Finally, the subtree rule (SM.1) is checked.

$$\frac{\begin{array}{c}\mathsf{M} = \mathsf{C}_0\ \mathsf{m}(\overline{\mathsf{C}\ \mathsf{x}})\ \{\ \mathsf{return}\ \mathsf{t}_0;\ \} \quad AT(\mathsf{M}) = \mathcal{A} \quad AT(\overline{\mathsf{C}\ \mathsf{x}}) \Rightarrow \mathcal{A} \\ \mathit{filter}(\overline{\mathsf{C}\ \mathsf{x}},\mathcal{A}) = \overline{\mathsf{D}\ \mathsf{y}} \quad \overline{\mathsf{y}} : \overline{\mathsf{D}}, \mathit{this} : C \vdash \mathsf{t}_0 : \mathsf{E}_0 \quad \mathsf{E}_0 <: \mathsf{C}_0 \\ CT(\mathsf{C}) = \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \{\ldots\} \quad \mathcal{A} \Rightarrow AT(\mathsf{C}_0) \\ \mathit{override}(\mathsf{m},\mathsf{D},\overline{\mathsf{C}} \to \mathsf{C}_0,\mathcal{A}) \quad AT(\overline{\mathsf{C}\ \mathsf{x}}) \Rightarrow AT(\overline{\mathsf{C}})\end{array}}{\mathsf{M}\ \mathsf{OK}\ \mathsf{in}\ \mathsf{C}}$$

7) Class Typing (C OK): The class typing rule appears very complex, because it covers several annotation rules. Each rule by itself is simple though. To distinguish the occurrences of $\overline{\mathsf{g}}$ as constructor parameters, super invocation parameters, and fields of the superclass, we also use $\overline{\mathsf{g}}$' and $\overline{\mathsf{g}}$" but still assume that all $\overline{\mathsf{g}}$'s are named the same ($\overline{\mathsf{g}} = \overline{\mathsf{g}}' = \overline{\mathsf{g}}''$). The same for $\overline{\mathsf{C}\ \mathsf{f}}$ that is used both for fields and constructor parameters ($\overline{\mathsf{C}\ \mathsf{f}} = \overline{\mathsf{C}\ \mathsf{f}}'$).

First, rule (L.1) checks that the super class is always present when the given class is present ($\mathcal{A} \Rightarrow AT(\mathsf{D})$). Second, rule (K.1) specifies that the super constructor call receives exactly those parameters from the constructor's parameter list that are defined as fields in the super class ($\mathcal{A} \Rightarrow (AT(\overline{\mathsf{D}\ \mathsf{g}}) \equiv AT(\overline{\mathsf{g}}') \equiv AT(\overline{\mathsf{D}\ \mathsf{g}}'')$), in which $\overline{\mathsf{D}\ \mathsf{g}}''$ is filtered by *filter*. Third, rule (K.2) specifies that the other constructor parameters match the field assignments ($AT(\overline{\mathsf{C}\ \mathsf{f}}) \equiv AT(\overline{\mathsf{this.f=f}})$) and that those match the fields declared in the class ($AT(\overline{\mathsf{this.f=f}}) \equiv AT(\overline{\mathsf{C}\ \mathsf{f}}')$).[4] Fourth, rule (K.3) specifies that all classes corresponding to types used in the constructor's parameter list must be present ($AT(\overline{\mathsf{C}\ \mathsf{f}}) \Rightarrow AT(\overline{\mathsf{C}})$ and $AT(\overline{\mathsf{D}\ \mathsf{g}}) \Rightarrow AT(\overline{\mathsf{D}})$). Finally, several subtree rules for fields, methods and constructors (SL.1–2,

[4]The implication $\mathcal{A} \Rightarrow \ldots$ is not needed for this rule because all elements share the same annotateable parent element, the class declaration.

SK.1–3) are checked here.

$$\frac{\begin{array}{c}\mathsf{K} \;=\; \mathsf{C}(\overline{\mathsf{D}\ \mathsf{g}},\ \overline{\mathsf{C}\ \mathsf{f}})\ \{\ \mathsf{super}(\bar{\mathsf{g}}');\ \overline{\mathsf{this.f=f;}}\ \}\quad \overline{\mathsf{M}}\ \mathsf{OK\ in\ C} \\ AT(\mathsf{C}) = \mathcal{A}\quad \mathit{filter}(\mathit{fields}(\mathsf{D}),\mathcal{A}) \;=\; \overline{\mathsf{D}\ \mathsf{g}''} \\ \mathcal{A} \Rightarrow AT(\mathsf{D})\quad AT(\overline{\mathsf{C}\ \mathsf{f}}) \equiv AT(\overline{\mathsf{this.f=f}}) \equiv AT(\overline{\mathsf{C}\ \mathsf{f}'}) \\ \mathcal{A} \Rightarrow (AT(\overline{\mathsf{D}\ \mathsf{g}}) \equiv AT(\bar{\mathsf{g}}') \equiv AT(\overline{\mathsf{D}\ \mathsf{g}''})) \\ AT(\overline{\mathsf{C}\ \mathsf{f}}) \Rightarrow AT(\overline{\mathsf{C}})\quad AT(\overline{\mathsf{D}\ \mathsf{g}}) \Rightarrow AT(\overline{\mathsf{D}}) \\ AT(\overline{\mathsf{C}\ \mathsf{f}'}) \Rightarrow \mathcal{A}\quad AT(\overline{\mathsf{M}}) \Rightarrow \mathcal{A}\quad AT(\overline{\mathsf{D}\ \mathsf{g}}) \Rightarrow \mathcal{A} \\ AT(\overline{\mathsf{C}\ \mathsf{f}}) \Rightarrow \mathcal{A}\quad AT(\bar{\mathsf{g}}') \Rightarrow \mathcal{A}\quad AT(\overline{\mathsf{this.f=f}}) \Rightarrow \mathcal{A}\end{array}}{\mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}\ \mathsf{f}}';\ \mathsf{K}\ \overline{\mathsf{M}}\ \}\ \mathsf{OK}}$$

8) SPL Typing (P OK): Finally, we can define what it means for an SPL to be well-typed. As in FJ, a CFJ program $(\overline{\mathsf{L}}, \mathsf{t}, \mathsf{FM})$ – i.e., an SPL – is well-typed if $\overline{\mathsf{L}}\ \mathsf{OK}$ and $\vdash \mathsf{t} : \mathsf{C}$ (and it does not contain stupid casts).

IV. Variant Generation

Programs written in CFJ are never directly evaluated. Thus, there are no evaluation rules for CFJ, and it is not possible or necessary to prove type-soundness. Instead with a valid feature selection, a well-typed FJ program is generated – by removing some annotated code fragments – that can be evaluated with FJ's evaluation rules (see [17]). For FJ, type-soundness has already been proved [17]. The interesting property is, whether every FJ program variant that is generated from a well-typed CFJ SPL is well-typed. In the following, we define the semantics of the generation process and analyze its properties.

A. Variant Generation Semantics

To generate a program variant, we define a function *variant* that takes a CFJ program P_{CFJ} and a feature selection $v \subseteq F$ as input and returns an FJ program. The function *variant* descends recursively through the SPL and checks all code fragments that can be annotated. To them a function *remove* is applied, that evaluates the annotations (variables in the propositional formulas are assigned with truth values: *true* if the feature is in v, *false* otherwise). Those code fragments, for which the annotation evaluates to *false* are removed, the remaining code fragments are stripped of their annotations.[5]

For brevity, we write $\mathit{variant}(\mathsf{x}, v)$ as $[\![\mathsf{x}]\!]$ and $\mathit{remove}(\bar{\mathsf{x}}, v)$ as $\langle\!\langle \bar{\mathsf{x}} \rangle\!\rangle$. The *variant* function is defined (syntax-driven) with the following generation rules:

$$[\![\mathsf{x}]\!] = \mathsf{x} \tag{G.1}$$

$$[\![\mathsf{t.f}]\!] = [\![\mathsf{t}]\!].\mathsf{f} \tag{G.2}$$

$$[\![\mathsf{t.m}(\bar{\mathsf{t}})]\!] = [\![\mathsf{t}]\!].\mathsf{m}([\![\langle\!\langle \bar{\mathsf{t}} \rangle\!\rangle]\!]) \tag{G.3}$$

$$[\![\mathsf{new\ C}(\bar{\mathsf{t}})]\!] = \mathsf{new\ C}([\![\langle\!\langle \bar{\mathsf{t}} \rangle\!\rangle]\!]) \tag{G.4}$$

$$[\![(\mathsf{C})\mathsf{t}]\!] = (\mathsf{C})[\![\mathsf{t}]\!] \tag{G.5}$$

$$[\![\mathsf{C\ m}(\overline{\mathsf{C}\ \mathsf{x}})\ \{\mathsf{return\ t;}\}]\!] = \mathsf{C\ m}(\langle\!\langle \overline{\mathsf{C}\ \mathsf{x}} \rangle\!\rangle)\ \{\mathsf{return}\ [\![\mathsf{t}]\!];\} \tag{G.6}$$

$$[\![(\overline{\mathsf{L}}, \mathsf{t}, \mathsf{FM})]\!] = ([\![\langle\!\langle \overline{\mathsf{L}} \rangle\!\rangle]\!], [\![\mathsf{t}]\!]) \tag{G.7}$$

$$[\![\mathsf{C}(\overline{\mathsf{C}\ \mathsf{f}})\ \{\mathsf{super}(\bar{\mathsf{f}});\ \overline{\mathsf{this.f=f;}}\}]\!] = \mathsf{C}(\langle\!\langle \overline{\mathsf{C}\ \mathsf{f}} \rangle\!\rangle)\ \{\mathsf{super}(\langle\!\langle \bar{\mathsf{f}} \rangle\!\rangle);\ \langle\!\langle \overline{\mathsf{this.f=f;}} \rangle\!\rangle\} \tag{G.8}$$

[5] In CIDE, *remove* is implemented using transformations of the abstract syntax tree, which ensures that separating tokens like commas between parameters are placed correctly [22].

$$[\![\mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}\ \mathsf{f}};\ \mathsf{K}\ \overline{\mathsf{M}}\ \}]\!] = \mathsf{class\ C\ extends\ D}\ \{\ \langle\!\langle \overline{\mathsf{C}\ \mathsf{f}} \rangle\!\rangle;\ [\![\mathsf{K}]\!]\ [\![\langle\!\langle \overline{\mathsf{M}} \rangle\!\rangle]\!]\ \} \tag{G.9}$$

B. Properties of Variant Generation

With the variant generation semantics, only optional code fragments can be removed, so that every generated FJ program is syntactically correct. The challenge remains to prove that *every* generated program variant is well-typed.

THEOREM IV.1 (*Variant generation preserves typing*): Every FJ program variant that is generated from a well-typed SPL P_{CFJ} with a valid feature selection v is well-typed.

$$\frac{P_{CFJ}\ \text{is well-typed}\quad v \subseteq F\quad \mathit{valid}(v)}{\mathit{variant}(P_{CFJ}, v)\ \text{is well-typed}}$$

Proof sketch: First, a well-typed CFJ program in that every annotation evaluates to *true*, i.e., no code is removed, is automatically a well-typed FJ program, because the syntax for FJ and CFJ is the same and CFJ's typing rules completely cover FJ's typing rules, they are only stricter.

For all further cases, where code fragments are removed, we induct over the generation rules (G.1–9). For each case, we analyze possible FJ typing rules that might be violated by a removal. We show here only an excerpt:

CASE (G.1): No removal, typing not affected.

[...]

CASE (G.9): Field and method declarations can be removed during generation. The subcases for typing rules T-VAR, T-FIELD, T-NEW, T-UCAST, T-DCAST, M OK in C, and C OK are not affected by a removed method declaration, i.e. none of these rules checks for the presence of a specific method. Still, a removed method can affect the FJ typing rule T-INVK. In this subcase, if a method is removed, T-INVK can fail to type a term $\mathsf{t}_0.\mathsf{m}(\bar{\mathsf{t}})$ in case *mtype* does not find the now removed method. However, this problem cannot occur in a program generated from a well-typed CFJ program. Already in CFJ, the T-INVK rule would fail because *mtype* called with the terms annotation $\mathcal{A} = AT(\mathsf{t}_0.\mathsf{m}(\bar{\mathsf{t}}))$ only returns methods that are present when the term itself is present. That is, method declarations that are removed by the *variant* function from a well-typed CFJ program cannot cause ill-typed program variants.

Because the proof is straightforward but highly repetitive, we skipped over the remaining cases for generation rules (G.2–8) and for removing field declarations in (G.9). In each case, a possible typing problem in the generated FJ program is already covered by a stricter typing rule in CFJ. ■

Interestingly, the mechanics of this proof are very similar to our discussion that lead to the annotations rules (L.1–T.5) initially. This means, such a proof can be used constructively, to extend the typing rules of another calculus, be it of some FJ extensions like FGJ [17] or FJI [18]; or of a larger Java calculus like Classic Java [13], Java$_{light}$ [25], or Java^s [12]; or even of a type system for completely different programming language.

V. Lessons Learned

Formalizing CFJ was a helpful experience, even beyond just theoretical insights, as we illustrate in this section.

When implementing annotation checks in CIDE, we implemented them on top of the Java compiler (used from Eclipse's Java framework). That is, we assume that every SPL is a well-typed Java program that additionally adheres to some conditions on its annotations. The CFJ calculus assured us that it is feasible to extend an existing type system with annotation checks. However, it also shows that some lookup functions required for these annotations checks need to be adapted (*mtype* and *overrides* in CFJ). Previously in CIDE, we checked a method invocation only against its direct target, we did not search for overridden methods that might still be present. Changing the lookup functions was an easy fix.

Furthermore, the small size of the calculus helped us analyzing some conditions more clearly and thoroughly than we did before in CIDE. First, it showed us possibilities to integrate domain constraints in the type-checks, which we did not earlier. Second, it revealed a subtle error. In earlier work, we stated *"Every parameter in a method call must have exactly the same [directly declared] colors as the parameter in the method declaration"* [22]. The direct translation for T-Invk would be $(\mathcal{A} \Rightarrow AT(\bar{\mathsf{t}})) \equiv (AT(\mathsf{M}) \Rightarrow AT(\overline{\mathsf{D}}))$,[6] which is actually incorrect and must be $\mathcal{A} \Rightarrow (AT(\bar{\mathsf{t}}) \equiv AT(\overline{\mathsf{D}}))$ instead, as discussed in Section III-E4. The handy size of the calculus allowed us to analyze these subtle differences in detail which eventually revealed our mistake.

When sketching the annotation checking mechanism for CIDE in earlier work, we were aware that we did not cover all conditions. We assumed that we might need *"more complex logic, rule, or specification languages"* [22] that are checked in one big step to handle language semantics of abstract methods or interfaces. To our surprise, all annotation checks in CIDE including overridden methods (and including all checks we found when discussing extensions of FJ with abstract methods and interfaces) could be modeled with implications on *pairs* (or triples) of code fragments. This reassures us of our 'small checks' approach with many small conditions instead of feeding many collected, possibly complex conditions in an SAT solver or theorem prover [16], [11], [30]. (We prefer this small checks approach, because it is able to point out several violations of the annotation rules at the same time, instead of only a single error reconstructed from an SAT solver's result.)

Finally, we found the proof of the type preservation theorem itself very interesting, as we found that we can use it constructively. Though a complete proof for a language like full Java or C# is too complex, the basic mechanisms of the proof provide a framework how to derive annotation rules. This helps us to improve CIDE's annotation checks for Java and to extend CIDE for new languages. It might even contribute to our longstanding vision of providing a *safe* and *language-independent* SPL tool.

[6] M represents the target method, $\mathcal{A}$, as usual, represents the annotation of the current method invocation term.

VI. Open Problem: Alternative Features

In the formalization of CFJ, CIDE's roots in decomposing legacy applications are clearly visible. It is possible to make code fragments optional and to express conditions like *either FeatureA or FeatureB must be selected* using domain constraints. However, it is difficult to have two alternative implementations of the same method (e.g., two different buffer strategies LFU and LRU in a database) while the rest of the program expects either implementation to be present – a functionality often used when constructing new SPLs.

Every CFJ class is a valid FJ class (is a valid Java class) and thus cannot have two methods with the same name. Further in CFJ and FJ, a method contains only a single term that cannot be annotated. The only way to have two alternative implementations of a method is to (ab)use overriding and declare one implementation in a subclass that is replaced by another annotated implementation in the superclass. Depending on the feature selection either the one or the other method can be removed and *mtype* ensures that at least one implementation is available in every variant in which this method is invoked.

There are different approaches how we could extend CFJ to handle alternative features, apart from abusing method overriding. First, we could allow multiple methods with the same name inside one CFJ class and make sure that in every generated variant at most one of these methods is present. Second, following the concept of *meta-expression* annotations (additionally to *presence conditions*) of Czarnecki and Antkiewicz [9], we could model alternative implementations as external annotations that are also checked by the typing system and evaluated during variant generation. Third, we could allow multiple return statements in a method and evaluate only the first one that is present (a solution we currently use as workaround with full Java in CIDE). All of these solutions have advantages and disadvantages (e.g., some make it impossible to reuse/extend the Java compiler), but their discussion and formalization would exceed the scope of the paper, and will be addressed in future work.

VII. Related Work

Our aim of proving type safety for an entire SPL instead of only for a single program is in line with three approaches from the community of generative programming.

First, an influential approach proposed by Huang et al. ensures correct language semantics for Java code generated by their tool SafeGen [16]. Though their tool is used for metaprogramming in general, not as SPL technology, the basic idea is similar to proving the variant generation process in CFJ safe. Using first-order logics and theorem provers, they check whether generators written in their confined meta-language (with selection and iteration operators) produce well-typed output for arbitrary Java input. However, checks cover only some language semantics of Java, i.e., there is no *guarantee* that the output is well-typed.

An approach to ensure language semantics of a programming language in all variants of an SPL called *safe composition* was proposed by Thaker et al. [30]. They analyze

language semantics of Jak, a Java dialect for feature-oriented programming [8]. To check SPL typing, they identify six constraints that need to be satisfied, which their program *safegen* maps to propositional formulas and checks with an SAT solver. One constraint deals with references to fields and methods (roughly corresponding to T-FIELD and T-INVK), two deal with abstract classes and interfaces (no correspondence in FJ), and three deal with specific constructs of the Jak composition mechanism (no correspondence in FJ). Their checks are not claimed or even proved complete, and in fact – compared to CFJ – checks that ensure the presence of types uses in signatures are missing, e.g., (M.1), (M.3).

Closest to our work is an SPL tool for models called *fmp2rsm* in which model elements are annotated with presence conditions very similar to our formalization [9]. Czarnecki and Pietroszek developed an approach to check language semantics of models in this tool against OCL constraints for all variants [11]. In MOF, language semantics for models – e.g., 'an association in UML class diagrams connects exactly two elements' – can be specified formally (and machine-readable) using OCL constraints. Their tool now transforms the presence conditions and OCL constraints into a propositional formula that can be solved by an off-the-shelf SAT solver in one step. Error messages are reconstructed from the SAT solver's result. Again, well-formedness can only be guaranteed against those constraints that have been specified (machine-readable) with OCL. For example, for UML those must be first inferred from the informal text of the UML specification, similar to how Java constraints must be inferred from the informal Java Language Specification. The authors do not discuss completeness of their inferred OCL constraints. Interestingly, their modeling tool also includes meta-expression annotations that allow alternative values for one node, however meta-expressions are not (yet) included in their well-formedness checks.

Our formalization is novel, because it shows how previous approaches can be extended to *guarantee* well-typedness not only for selected constraints but for all language semantics. For implementation, we developed an extension to our existing tool CIDE, but we could as well reuse the implementation from *safegen* or *fmp2rsm* and encode the results of our calculus as either conditions suitable for *safegen* or as OCL constraints to be checked by *fmp2rsm*. All approaches share the same overall goal of preventing errors in the complex task of SPL development early on.

Finally, in a parallel line of research we developed an algebra [4] and two calculi (incl. operational semantics and type system) [2], [3] for feature composition. Instead of annotating one integrated code base as in CIDE and *fmp2rsm*, this work aims at composing different artifacts that are physically separated in different files. Especially the calculus *Feature Featherweight Java (FFJ)* [3] is close to our work, because it is also based on FJ. FFJ extends syntax, typing, and evaluation rules for new language constructs; checks equivalent to [30] and ours are directly integrated into the typing rules. FFJ is proved type-sound using the standard preservation and progress theorems, whereas CFJ generates FJ code and only needs to prove the generation process. In some sense, feature composition and our approach of feature annotations are two sides of the same coin: one removes code from a common base, the other merges already separated code. Both approaches can be used to implement SPLs.

VIII. Conclusion

In this paper, we have formally discussed CFJ, a calculus for an SPL based on Featherweight Java. With CFJ we can prove that – if the SPL is well-typed – all possible program variants are well-typed as well, without generating and compiling them first. We have shown that CFJ can be modeled on top of FJ, extending only the typing rules and auxiliary functions with implications on pairs of annotations.

The formalization was motivated by our SPL tool CIDE for Java and other languages. Though, CFJ (or FJ) covers only a small excerpt from the Java specification, the formalization confirmed us in our approach to check annotations on top of an existing Java compiler and at the same time pointed out some subtle errors. Finally, the proof that variant generation preserves typing can also be used constructively, to design an SPL type system for a new annotated language.

In future work, we intend to explore paths toward handling alternative features and extensions of other code and non-code languages and their interactions (inter-language semantics). Our long term goal is to provide a language-independent SPL tool that counteracts the inherent complexity of SPLs by detecting possible errors as early as possible.

References

[1] S. Apel, T. Leich, and G. Saake, "Aspectual feature modules," *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 162–180, 2008.

[2] S. Apel and D. Hutchins, "An overview of the gDeep calculus," Department of Informatics and Mathematics, University of Passau, Germany, Tech. Rep. MIP-0712, 2007.

[3] S. Apel, C. Kästner, and C. Lengauer, "An overview of Feature Featherweight Java," Department of Informatics and Mathematics, University of Passau, Germany, Tech. Rep. MIP-0802, Apr. 2008.

[4] S. Apel, C. Lengauer, B. Möller, and C. Kästner, "An algebra for features and feature composition," in *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology (AMAST)*, ser. Lecture Notes in Computer Science, vol. 5140. Springer-Verlag, Jul. 2008, pp. 36–50.

[5] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Boston, MA, USA: Addison-Wesley, 1998.

[6] D. Batory, "Feature models, grammars, and propositional formulas." in *Proc. Int'l Software Product Line Conference*, Sep. 2005, pp. 7–20.

[7] D. Batory and B. J. Geraci, "Composition validation and subjectivity in GenVoca generators," *IEEE Trans. Softw. Eng.*, vol. 23, no. 2, pp. 67–82, 1997.

[8] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 355–371, 2004.

[9] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants," in *Proc. Int'l Conf. Generative Programming and Component Engineering*, 2005, pp. 422–437.

[10] K. Czarnecki and U. Eisenecker, *Generative programming: methods, tools, and applications*. ACM Press, 2000.

[11] K. Czarnecki and K. Pietroszek, "Verifying feature-based model templates against well-formedness OCL constraints," in *Proc. Int'l Conf. Generative Programming and Component Engineering*. New York, NY, USA: ACM Press, 2006, pp. 211–220.

[12] S. Drossopoulou, T. Valkevych, and S. Eisenbach, "Java type soundness revisited," Department of Computing, Imperial College London, Tech. Rep. 09, 2000.

[13] M. Flatt, S. Krishnamurthi, and M. Felleisen, "Classes and mixins," in *Proc. Conf. Principles of Programming Languages*. New York, NY, USA: ACM, 1998, pp. 171–183.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java™ Language Specification*, 3rd ed., ser. The Java™ Series. Addison-Wesley Professional, 2005.

[15] M. L. Griss, "Implementing product-line features by composing aspects," in *Proc. Int'l Software Product Line Conference*. Norwell, MA, USA: Kluwer Academic Publishers, 2000, pp. 271–288.

[16] S. Huang, D. Zook, and Y. Smaragdakis, "Statically safe program generation with SafeGen," in *Proc. Int'l Conf. Generative Programming and Component Engineering*. Springer, 2005, pp. 309–326.

[17] A. Igarashi, B. Pierce, and P. Wadler, "Featherweight Java: A minimal core calculus for Java and GJ," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 3, pp. 396–450, 2001.

[18] A. Igarashi and B. C. Pierce, "On inner classes," *Inf. Comput.*, vol. 177, no. 1, pp. 56–89, 2002.

[19] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang, "XVCL: XML-based variant configuration language," in *Proc. Int'l Conf. on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society, 2003, pp. 810–811.

[20] K. Kang *et al.*, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, Nov. 1990.

[21] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *Proc. Int'l Conf. on Software Engineering*. New York, NY, USA: ACM, May 2008, pp. 311–320.

[22] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory, "Language-independent safe decomposition of legacy applications into features," School of Computer Science, University of Magdeburg, Germany, Tech. Rep. 2, Mar. 2008.

[23] C. Krueger, "Easing the transition to software mass customization," in *Proc. Int'l Workshop on Software Product-Family Eng.* London, UK: Springer-Verlag, 2002, pp. 282–293.

[24] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," in *Proc. Int'l Conf. on Software Engineering*, 2006, pp. 112–121.

[25] T. Nipkow and D. von Oheimb, "Java light is type-safe – definitely," in *Proc. Conf. Principles of Programming Languages*. New York, NY, USA: ACM, 1998, pp. 161–170.

[26] N. Nystrom, S. Chong, and A. Myers, "Scalable extensibility via nested inheritance," in *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications*. New York, NY, USA: ACM Press, 2004, pp. 99–115.

[27] B. C. Pierce, *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002.

[28] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer, 2005.

[29] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[30] S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe composition of product lines," in *Proc. Int'l Conf. Generative Programming and Component Engineering*. New York, NY, USA: ACM, 2007, pp. 95–104.