Compositional Type-Checking for Delta-Oriented Programming *

Ina Schaefer[1] † Lorenzo Bettini[2] Ferruccio Damiani[2]

[1]Chalmers University of Technology, 421 96 Gothenburg, Sweden
[2]Dipartimento di Informatica, Università di Torino, C.so Svizzera, 185 - 10149 Torino, Italy

Abstract

Delta-oriented programming is a compositional approach to flexibly implementing software product lines. A product line is represented by a *code base* and a *product line declaration*. The code base consists of a set of delta modules specifying modifications to object-oriented programs. The product line declaration provides the connection of the delta modules with the product features. This separation increases the reusability of delta modules. In this paper, we provide a foundation for compositional type checking of delta-oriented product lines of JAVA programs by presenting a minimal core calculus for delta-oriented programming. The calculus is equipped with a constraint-based type system that allows analyzing each delta module in isolation, such that that also the results of the analysis can be reused. By combining the analysis results for the delta modules with the product line declaration it is possible to establish that all the products of the product line are well-typed according to the JAVA type system.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.3 [*Studies of Program Constructs*]: Type Structure

General Terms Design, Languages, Theory

Keywords Java, Software Product Line, Type System

1. Introduction

Delta-oriented programming (DOP) [29, 31] is a flexible compositional approach for implementing software product lines [10]. The implementation of a product line in DOP is organized into a *code base* and a *product line declaration*. The code base consists of a set of the delta modules that comprise modifications of object-oriented programs. A delta module can add classes, remove classes or modify classes by changing the class structure. The product line declaration provides the connection between the delta modules and the variabilities of the products defined in terms of product features [18]. For each delta module, an application condition over the product features is specified, and an application ordering for the delta modules is fixed. The separation between delta modules and product line declaration increases the reusability of delta modules, making it possible to develop different product lines by re-using the same delta modules.

Delta-oriented programming is an extension of *feature-oriented programming* (FOP) [6], a compositional approach for implementing software product lines (see [31] for a straightforward embedding of FOP into DOP). The code base of a feature-oriented product line contains a set of feature modules that correspond directly to product features. Hence, the product line declaration for a feature-oriented product line only provides the set of valid feature configurations and a composition ordering of the feature modules. A feature module can be understood as a delta module without remove operations such that product line development always starts from base feature modules comprising the mandatory product features. In DOP, any product can be chosen as a base (delta) module. Hence, DOP supports proactive product line development, where all possible products are planned in advance, as well as extractive product line development [24] which starts from existing legacy product implementations. Moreover, the application conditions associated with delta modules allow handling combinations of features explicitly. This provides an elegant way to counter the optional-feature problem [21] where two optional features require additional glue code to cooperate properly. However, the additional flexibility provided by DOP makes it challenging to ensure that for every feature configuration a unique product can be generated and that all generated products are well-typed.

In this paper, we provide a foundation for compositional type checking for DOP by considering IFΔJ (IMPERATIVE FEATHERWEIGHT DELTA JAVA), a core calculus for DOP product lines of JAVA programs. IFΔJ is based on an imperative variant of FJ (FEATHERWEIGHT JAVA) [16] that is used to implement the products. The calculus is equipped with a constraint-based type system that allows analyzing each delta module in isolation, such that the analysis results can be reused for different product lines like the delta modules. By combining the analysis results of the delta modules with the product line declaration, it is possible the establish that all the products of the product line are well-typed according to the JAVA type system.

The paper is organized as follows. Section 2 introduces DOP by an example. Section 3 discusses the requirements in designing a type system for DOP in the relation with existing type systems for FOP. Section 4 explains the underlying calculus for implementing products. Section 5 presents the syntax and semantics of IFΔJ. Sections 6 and 7 describe the constraint-based type system. A comparison with aspect-oriented programming is provided in Section 8.

* Partially supported by the German-Italian University Centre (Vigoni program) and by MIUR (PRIN 2009 DISCO).

† This author has been supported by the Deutsche Forschungsgemeinschaft (DFG) and the EU project FP7-231620 HATS.

AOSD '11 March 21-25, Porto de Galinhas, Pernambuco

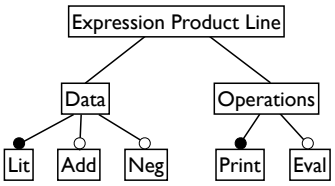

Figure 1. Feature model for Expression Product Line

Related approaches and future work are discussed in Sections 9 and 10. Proof sketches of the main results are available in [30].

2. Delta-oriented Programming

In order to illustrate the main concepts of delta-oriented programming as introduced in [31], we use a variant the *expression product line* (EPL) as described in [26]. The EPL is based on the *expression problem* [36], an extensibility problem, that has been proposed as a benchmark for data abstractions capable to support new data representations and operations. We consider the following grammar:

Exp ::= Lit | Add | Neg Lit ::= <non−negative integers>
Add ::= Exp "+" Exp Neg ::= "−" Exp

Two different operations can be performed on the expressions described by this grammar: printing, which returns the expression as a string, and evaluating, which returns the value of the expression. The products in the EPL can be described by two feature sets, the ones concerned with data Lit, Add, Neg and the ones concerned with operations Eval and Print. Lit and Print are mandatory features. The features Add, Neg and Eval are optional. Figure 1 shows the feature model of the EPL.

The example aims at illustrating the constructs of the IFΔJ calculus presented in Section 5, rather than to provide an elegant implementation of the EPL. Although the example uses a more general syntax (including the primitive type `int`, the shortcut syntax for operations on strings, and the sequential composition) than the syntax of the IFJ calculus presented in Section 4, the encoding in IFJ is straightforward. We refer to [31] for examples of DOP programming of the EPL that exploit the full JAVA syntax.

Delta Modules The main concept of DOP are delta modules which are containers of modifications to an object-oriented program. The modifications may add, remove or modify classes. Modifying a class means to change the super class, to add or to remove fields or methods or to modify methods. The modification of a method can either replace the method body by another implementation, or wrap the existing method using the `original` construct (similar to the `Super()` call in AHEAD [6]). The `original` construct expresses a call to the method with the same name before the modifications and is bound at the time the product is generated. Before or after the `original` construct, other statements can be introduced wrapping the existing method implementation. As pointed out in [31], DOP supports extractive product line development [24], starting for existing legacy products. Listing 1 contains a delta module for introducing an existing legacy product, realizing the features Lit, Add and Print. Listing 2 contains the delta modules for adding the evaluation functionality to the classes `Lit` and `Add`. Listing 3 contains the delta modules for incorporating the Neg feature by adding and modifying the class `Neg` and for adding glue code required by the two optional features Add and Neg to cooperate properly. Listing 4 contains the delta module for removing the Add feature from the legacy product.

Delta-oriented Product Lines A delta-oriented product line consists of a *code base* and a *product line declaration*. The code base

```
delta DLitAddPrint{
  adds class Exp extends Object { // only used as a type
     String toString() { return ""; }
  }
  adds class Lit extends Exp {
    int value;
    Lit setLit(int n) { value = n; return this; }
    String toString() { return value + ""; }
  }
  adds class Add extends Exp {
    Exp expr1;
    Exp expr2;
    Add setAdd(Exp a, Exp b) { expr1 = a; expr2 = b; return this; }
    String toString() { return  expr1.toString() + " + " + expr2.toString(); }
  }
}
```

Listing 1: Delta module introducing a legacy product

```
delta DLitEval {
  modifies Exp { adds int eval() { return 0; } }
  modifies Lit { adds int eval() { return value; } }
}

delta DAddEval {
  modifies Add { adds int eval() { return expr1.eval() + expr2.eval(); } }
}
```

Listing 2: Delta modules for the Eval feature

```
delta DNeg  {
  adds class Neg extends Exp {
    Exp expr;
    Neg setNeg(Exp a) { expr = a; return this; }
  }
}

delta DNegPrint {
  modifies Neg { adds String toString() { return "-" +  expr.toString(); } }
}

delta DNegEval{
  modifies Neg { adds int eval() { return (−1) ∗ expr.eval(); } }
}

delta DAddNegPrint {
  modifies Add { modifies toString { return "(" +  original + ")"; } }
}
```

Listing 3: Delta modules for Neg, Print and Eval features

```
delta DremAdd { removes Add }
```

Listing 4: Delta module removing the Add feature

consists of a set of delta modules, while the product line declaration creates the connection to the product line variability specified in terms of product features. An application condition is attached to each delta module in a `when` clause specifying by a propositional constraint over the set of features for which feature configurations the delta module has to be applied. Since only feature configurations that are valid according to the feature model are used for product generation, the application conditions are understood as a conjunction with the formula describing the set of valid fea-

```
features Lit, Add, Neg, Print, Eval
configurations Lit & Print
deltas
  [ DLitAddPrint,
    DNeg when Neg ]

  [ DLitEval when Eval,
    DNegPrint when Neg,
    DNegEval when (Neg & Eval),
    DremAdd when !Add ]

  [ DAddEval when (Add & Eval),
    DAddNegPrint when (Add & Neg) ]
```

Listing 5: Specification of the EPL

ture configurations.[1] Additionally, an application order of the delta modules is fixed by defining a total order on a partition of the set of delta modules. Deltas in the same part can be applied in any order to the previous product, but the order of the parts is fixed. The ordering captures semantic requires-relations that are necessary for the applicability of the delta modules. Listing 5 shows a product line declaration for the EPL. The order of delta module application is defined by an ordered list of the delta module sets which are enclosed by `[ .. ]`.

In order to obtain a product for a particular feature configuration, the modifications specified in the delta modules with valid application conditions are applied incrementally to the previously generated product. The first delta module is applied to the empty product. The modifications of a delta model are applicable to a (possibly empty) product if each class to be removed or modified exists and, for every modified class, if each method or field to be removed exists, if each method to be modified exists and has the same header as the modified method, and if each class, method or field to be added does not exist. During the generation of a product, every delta module must be applicable. Otherwise, the generation of the product fails. In particular, the first delta module that is applied can only contain additions.

Product Generation The generation of a product for a given feature configuration consists of two steps, performed automatically: (i) Find all delta modules with a valid application condition; and (ii) Apply the selected delta modules to the empty product in any linear ordering that respects the total order on the partition of the delta modules. If two delta modules add, remove or modify the same class, the ordering in which the delta modules are applied may influence the resulting product. However, for a product line implementation, it is essential to guarantee that for every valid feature configuration exactly one product is generated. This property is called *unambiguity* of the product line. A sufficient condition for unambiguity is that each part in the partition of the set of delta modules is *consistent*, that is, if one delta module in a part adds or removes a class, no other delta module in the same part may add, remove or modify the same class, and the modifications of the same class in different delta modules in the same part have to be disjoint. Defining the order of delta module application by a total ordering on a delta module partition provides an efficient way to ensure unambiguity, since only the consistency within each part has to be checked. Listing 6 depicts the product generated when the Lit, Neg, Print and Eval features are selected.

[1] In the examples, the valid feature configurations are represented by a propositional formula over the set of features. We refer to [5] for a discussion on other possible representations.

```
class Exp extends Object { String toString() { return ""; } int eval() { return 0; }
}
class Lit extends Exp {
  int value;
  Lit setLit(int n) { value = n; return this; }
  String toString() { return value + ""; }
  int eval() { return value; }
}
class Neg extends Exp {
  Exp expr;
  Net setNeg(Exp a) { expr = a; return this; }
  String toString() { return "-" + expr.toString(); }
  int eval() { return (−1) ∗ expr.eval(); }
}
```

Listing 6: Generated code for Lit, Neg, Print and Eval features

3. Compositional Type-checking of Product Lines

In this section, we explain design requirements for type systems for DOP/FOP product lines. We show how the type system proposed in [12] for FOP and the compositional type system for DOP presented in this paper realize these requirements.

Design Requirements for Type Systems for FOP and DOP An SPL is *type safe* if all valid products can be generated and are well-typed programs according to the type system of the target programming language. The type safety of an SPL could be checked by generating all products and type checking each separately. However, this direct approach has the drawback that it would be harder for the programmer to understand where the actual error in the code of the delta modules occurs based on an error raised while type checking a single product. Therefore, a first requirement for a product line type system is that the type safety of an SPL can be ensured without having to generate and inspect the code of all possible products.

The separation between the product line code base and product line declaration supported by DOP/FOP increases the reusability of the delta/feature modules, allowing the development of different product lines by re-using the same delta/feature modules. Hence, a second requirement is to design type systems that enable analyzing each delta/feature module in isolation. This makes the analysis results robust against changes of the product line and allows the re-use of the analysis results for different product lines such as the delta/feature modules.

Compositional type-checking of FOP product lines The calculus LIGHTWEIGHT FEATURE JAVA (LFJ) [12], based on LJ (LIGHTWEIGHT JAVA) [33] provides a formalization of FOP [6] together with a constraint-based type system that satisfies the two design requirements illustrate above. The approach consists of three steps.

1. A constraint-based type system for LJ that infers a set of constraints for a given LJ program. The constraints can be checked against the program in order to establish whether the program is well-typed according to the standard LJ type system.
2. A constraint-based type systems for LFJ that analyzes each feature module in isolation and infers a set of constraints for each feature module. The inferred constraints are divided into *structural constraints* (constraints of the same form as in the LJ constraint-based type system) and *composition and uniqueness constraints* that are imposed by the introduction and refinement operations of the feature modules. The constraints can be checked against the set of feature modules corresponding to a valid feature configuration in order to establish whether: (i) product generation succeeds, and (ii) the corresponding product is a well-typed LJ type program. Successful product generation requires that the classes/methods/fields introduced by a feature

module are not introduced by another feature module earlier in the composition and that the classes and methods refined by a feature module are introduced by another feature module earlier in the composition.

3. A procedure for translating the product line declaration and the constraints inferred for the feature modules to propositional formulas from which a formula is constructed whose satisfiability implies the type safety of the whole product line.

Checking the type safety of the product line by relying directly on the constraints inferred for the feature modules (at step 2) requires an explicit iteration on the valid feature configurations. So it is linear in the number of the products (which may be exponential in the number of features). Also checking the satisfiability of the propositional formula (at step 3) may be exponential in the number of features. However, the structure of the generated formula is suitable for fast analysis by modern SAT solvers and has been shown to scale well in practice [12]. If the formula is not satisfiable, it is not straightforward to trace the error back to the feature module that causes the error.

Compositional type-checking of DOP product lines The approach for ensuring FOP product-line type safety described above seems not straightforwardly applicable to DOP. The main issues are the flexible association between delta modules and features (provided by the `when` clauses) and the class/method/field removal operations that are not supported in FOP. Therefore, we develop a compositional type system for delta-oriented product lines that satisfies the design requirements illustrated above and relies on an abstract interpretation of product generation. The concepts of this approach are demonstrated for IFΔJ, a core calculus for delta-oriented product lines of JAVA programs based on IFJ (IMPERATIVE FEATHERWEIGHT JAVA) [7], an imperative version of FJ [16].

1. The first step of the approach is essentially the same as for LFJ described above. IFJ is equipped with a constraint-based type system that infers a set of *class constraints* $\mathscr{C}$ for an IFJ program `CT`. An IFJ program consists of a *class table* `CT`, i.e., a mapping from class names to class definitions. The inferred constraints are checked against the *class signature table* of `CT` in order to establish whether `CT` is a well-typed IFJ program. The class signature table is an abstract representation of the program without method bodies. The pair $\langle \mathit{signature}(\mathtt{CT}), \mathscr{C} \rangle$ suffices to establish whether `CT` is type safe.
2. The second step of our approach is quite different from that of LFJ. We define an abstraction of a delta module δ to consist of a pair $\langle \mathit{signature}(\delta), \mathscr{D}_\delta \rangle$, where $\mathit{signature}(\delta)$ is the *delta module signature* and $\mathscr{D}_\delta$ is a set of *delta clause-constraints*. The signature of a delta module is an abstract representation of the delta module without method bodies (the analogue of the class signature table) and can be constructed only from the delta module. The delta clause-constraints are inferred by a constraint-based type system for IFΔJ that analyzes each delta module δ separately. The generation of the delta module abstractions by considering each delta module in isolation satisfies the second requirement. Hence, the abstractions can be used (like the associated delta modules) across different product lines. The abstract representation of the delta modules suffices to construct the abstract representations of the possible products that are required to ensure their type safety. For each product, the class signature table can be generated by abstract product generation using the signatures of the delta modules that would be used to generate the product. The generation succeeds if and only if the generation of the corresponding product would succeed. The set of class constraints of the product can be generated only from the sets of delta clause-constraints of the corresponding delta modules. Therefore, type safety of a product line can be established only from the delta module signatures, the delta clause-constraints and the product line declaration without generating and inspecting the code of the products such that the first requirement is also satisfied.

CD	::=	`class C extends C {` $\overline{\mathtt{FD}}$; $\overline{\mathtt{MD}}$ `}`	classes
FD	::=	`C f`	fields
MD	::=	`C m (`$\bar{\mathtt{C}}$ $\bar{\mathtt{x}}$`){return e;}`	methods
e	::=	`x` | `e.f` | `e.m(`$\bar{\mathtt{e}}$`)` | `new C()` | `(C)e` | `e.f` $= \mathtt{e}$ | `null`	expressions

Figure 2. IFJ: syntax of classes

Due to the additional flexibility provided by DOP, there is currently no analogous of the third step of FOP type-checking (see item 3 above) for DOP type-checking. Therefore, checking type safety of a DOP product line is linear in the number of its products (which may be exponential in the number of features). We expect that generating program abstractions and performing the associated checks will take less time than generating the implementations of all products and checking them by a JAVA compiler. Furthermore, the constraints collected by the IFΔJ type system can be exploited to determine the exact location in a delta module causing a type error. The idea (not formalized in current presentation of the type system) is to record the location of the associated code in the delta modules for each generated constraint.

4. Imperative Featherweight Java

In this section, we introduce the syntax and the type system of IFJ (IMPERATIVE FEATHERWEIGHT JAVA) [7], a minimal imperative calculus for JAVA that we use as the underlying calculus to implement single products. IFJ is a variant of FJ (FEATHERWEIGHT JAVA) [16] that supports a more flexible initialization of fields (by field assignment expressions). This makes IFJ more suitable than FJ for the formalization of SPLs of JAVA programs, since (as already pointed out in [12]) the fact that FJ requires all the fields to be initialized in a single constructor call, whose parameter have to match the fields, makes it difficult to deal with product transformations that add (or remove) fields.

IFJ *Syntax* The abstract syntax of the IFJ constructs is given in Figure 2. Following [16], we use the overline notation for possibly empty sequences. For instance, we write "$\bar{\mathtt{e}}$" as short for a possibly empty sequence of expressions "$\mathtt{e}_1, \ldots, \mathtt{e}_n$" and "$\overline{\mathtt{MD}}$" as short for a possibly empty sequence of method definitions "$\mathtt{MD}_1 \ldots \mathtt{MD}_n$" (without commas). The empty sequence is denoted by $\bullet$. The length of a sequence $\bar{\mathtt{e}}$ is denoted by $\#(\bar{\mathtt{e}})$. We abbreviate operations on sequences of pairs in similar way, e.g., we write "$\bar{\mathtt{C}}\ \bar{\mathtt{f}}$" as short for "$\mathtt{C}_1\ \mathtt{f}_1, \ldots, \mathtt{C}_n\ \mathtt{f}_n$" and "$\bar{\mathtt{C}}\ \bar{\mathtt{f}}$;" as short for "$\mathtt{C}_1\ \mathtt{f}_1; \ldots \mathtt{C}_n\ \mathtt{f}_n$;". Sequences of named elements (field, method or parameter names, field, method or class definitions,..) are assumed to contain no duplicate names (that is, the names of the elements of the sequence must be distinct). The set of variables includes the special variable `this` (implicitly bound in any method declaration), which cannot be used as the name of a method's formal parameter.

A class definition `class C extends D {` $\overline{\mathtt{FD}}$; $\overline{\mathtt{MD}}$ `}` consists of its name `C`, its superclass `D` (which must always be specified, even if it is `Object`), a list of field definitions $\overline{\mathtt{FD}}$ and a list of method definitions $\overline{\mathtt{MD}}$. The fields declared in `C` are added to the ones declared by `D` and its superclasses and are assumed to have distinct names (i.e., there is no field shadowing). All fields and methods are public. Each class is assumed to have an implicit constructor that initializes all instance variables to `null`.

A class table CT is a mapping from class names to class definitions. The subtyping relation $<:$ on classes (types) is the reflexive and transitive closure of the immediate `extends` relation (the immediate subclass relation, given by the `extends` clauses in CT). The class `Object` has no members and its definition does not appear in CT. We assume that a class table CT satisfies the following sanity conditions: (*i*) $\mathtt{CT}(\mathtt{C}) = \mathtt{class\ C}\ \ldots$ for every $\mathtt{C} \in \mathit{dom}(\mathtt{CT})$ (*ii*) for every class name C (except `Object`) appearing anywhere in CT, we have $\mathtt{C} \in \mathit{dom}(\mathtt{CT})$; (*iii*) there are no cycles in the transitive closure of the immediate `extends` relation.

An IFJ program is a class table CT.[2] A class definition CD can be understood as a mapping from the keyword `extends` to a class name and from field/method names to field/method definitions. We use the metavariable a to range over field/method names, and the metavariable AD to range over field/method definitions. The lookup of the definition of a field/method a in class C is denoted by $\mathit{aDef}(\mathtt{C})(\mathtt{a})$. For every class C in $\mathit{dom}(\mathtt{CT})$, the function $\mathit{aDef}(\mathtt{C})$ is defined as follows:

$$\mathit{aDef}(\mathtt{C})(\mathtt{a}) = \begin{cases} \mathtt{CT}(\mathtt{C})(\mathtt{a}) & \text{if } \mathtt{a} \in \mathit{dom}(\mathtt{CT}(\mathtt{C})) \\ \mathit{aDef}(\mathtt{D})(\mathtt{a}) & \text{if } \mathtt{a} \notin \mathit{dom}(\mathtt{CT}(\mathtt{C})) \\ & \quad \text{and } \mathtt{CT}(\mathtt{C})(\mathtt{extends}) = \mathtt{D} \end{cases}$$

Given a field definition $\mathtt{FD} = \mathtt{C\ f}$ and a method definition $\mathtt{MD} = \mathtt{C\ m}(\bar{\mathtt{C}}\ \bar{\mathtt{x}})\{\cdots\}$, we write $\mathit{signature}(\mathtt{FD})$ to denote the type C of the field f and $\mathit{signature}(\mathtt{MD})$ to denote the type $\bar{\mathtt{C}} \to \mathtt{C}$ of the method m.

IFJ *Typing* A *class signature* CS is a class definition deprived of the bodies of its methods. The abstract syntax is as follows:

CS	**::=**	$\mathtt{class\ C\ extends\ C}\ \{\ \overline{\mathtt{FD}};\ \overline{\mathtt{MH}}\ \}$	class signatures
MH	**::=**	$\mathtt{C\ m}\ (\bar{\mathtt{C}}\ \bar{\mathtt{x}})$	method headers

A *class signature table* CST is a mapping from class names to class signatures. We write $\mathit{signature}(\mathtt{CT})$ to denote the class signature table consisting of the signatures of the classes in the class table CT. The lookup of the type of a field/method a in the signature of the class C is denoted by $\mathit{aType}(\mathtt{C})(\mathtt{a})$. For every class C in $\mathit{dom}(\mathtt{CST})$, the function $\mathit{aType}(\mathtt{C})$ is defined as follows:

$$\mathit{aType}(\mathtt{C})(\mathtt{a}) = \begin{cases} \mathtt{CST}(\mathtt{C})(\mathtt{a}) & \text{if } \mathtt{a} \in \mathit{dom}(\mathtt{CST}(\mathtt{C})) \\ \mathit{aType}(\mathtt{D})(\mathtt{a}) & \text{if } \mathtt{a} \notin \mathit{dom}(\mathtt{CST}(\mathtt{C})) \\ & \quad \text{and } \mathtt{CST}(\mathtt{C})(\mathtt{extends}) = \mathtt{D} \end{cases}$$

The subtyping relation $<:$ can be read off from the class signature table such that it is possible to check whether there are no cycles in the transitive closure of the `extends` relation. Moreover, by inspecting a class signature table, it is possible to check, for every class C in $\mathit{dom}(\mathtt{CST})$, whether the names of the fields defined in C are distinct from the names of the fields inherited from its superclasses, and whether the type of each method defined in C is equal to the type of any method with the same name defined in any of the superclasses of C. Therefore, in the following we can safely assume that a class signature table satisfies the following sanity conditions:

(*i*) $\mathtt{CS}(\mathtt{C}) = \mathtt{class\ C}\ \ldots$ for every $\mathtt{C} \in \mathit{dom}(\mathtt{CS})$;

(*ii*) for every class name C (except `Object`) appearing anywhere in CS, we have $\mathtt{C} \in \mathit{dom}(\mathtt{CS})$;

(*iii*) the transitive closure of the immediate `extends` relation is acyclic;

(*iv*) $\mathtt{C}_1 <: \mathtt{C}_2$ implies that, for all method names m, if $\mathit{aType}(\mathtt{C}_2)(\mathtt{m})$ is defined then $\mathit{aType}(\mathtt{C}_1)(\mathtt{m}) = \mathit{aType}(\mathtt{C}_2)(\mathtt{m})$; and

(*v*) $\mathtt{C}_1 <: \mathtt{C}_2$ and $\mathtt{C}_1 \neq \mathtt{C}_2$ imply that, for all field names f, if $\mathtt{f} \in \mathit{dom}(\mathtt{CST}(\mathtt{C}_2))$ then $\mathtt{f} \notin \mathit{dom}(\mathtt{CST}(\mathtt{C}_1))$.

[2] In FJ [16], a program is a pair $(\mathtt{CT}, \mathtt{e})$ of a class table and an expression e. We can encode it by adding to CT a class `class Main { C main() { return e; } }`, where C is the type of e.

Expression typing:

$$\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \quad (\text{T-VAR}) \qquad \Gamma \vdash \mathtt{null} : \bot \quad (\text{T-NULL})$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{C} \qquad \mathit{aType}(\mathtt{C})(\mathtt{f}) = \mathtt{A}}{\Gamma \vdash \mathtt{e.f} : \mathtt{A}} \quad (\text{T-FIELD})$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{aType}(\mathtt{C}_0)(\mathtt{m}) = \mathtt{A}_1 \cdots \mathtt{A}_n \to \mathtt{B} \qquad \forall i \in 1..n, \quad \Gamma \vdash \mathtt{e}_i : \mathtt{T}_i \quad \mathtt{T}_i <: \mathtt{A}_i}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\bar{\mathtt{e}}) : \mathtt{B}} \quad (\text{T-INVK})$$

$$\frac{\mathtt{C} \in \mathit{dom}(\mathtt{CST})}{\Gamma \vdash \mathtt{new\ C}() : \mathtt{C}} \quad (\text{T-NEW})$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \qquad \mathtt{T} <: \mathtt{C}}{\Gamma \vdash (\mathtt{C})\mathtt{e} : \mathtt{C}} \quad (\text{T-UCAST})$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \qquad \mathtt{C} <: \mathtt{T} \qquad \mathtt{C} \neq \mathtt{T}}{\Gamma \vdash (\mathtt{C})\mathtt{e} : \mathtt{C}} \quad (\text{T-DCAST})$$

$$\frac{\Gamma \vdash \mathtt{e}_0.\mathtt{f} : \mathtt{C} \quad \Gamma \vdash \mathtt{e}_1 : \mathtt{T} \quad \mathtt{T} <: \mathtt{C}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f} = \mathtt{e}_1 : \mathtt{C}} \quad (\text{T-ASSIG})$$

Method definition typing:

$$\frac{\mathtt{this} : \mathtt{C}, \bar{\mathtt{x}} : \bar{\mathtt{A}} \vdash \mathtt{e} : \mathtt{T} \qquad \mathtt{T} <: \mathtt{B}}{\mathtt{this} : \mathtt{C} \vdash \mathtt{B\ m}\ (\bar{\mathtt{A}}\ \bar{\mathtt{x}})\{\mathtt{return\ e;}\}\ \mathrm{OK}} \quad (\text{T-METHOD})$$

Class definition typing:

$$\frac{\mathtt{this} : \mathtt{C} \vdash \overline{\mathtt{MD}}\ \mathrm{OK}}{\vdash \mathtt{class\ C\ extends\ D}\ \{\ \overline{\mathtt{FD}};\ \overline{\mathtt{MD}}\ \}\ \mathrm{OK}} \quad (\text{T-CLASS})$$

Program typing:

$$\frac{\forall \mathtt{C} \in \mathit{dom}(\mathtt{CT}), \quad \vdash \mathtt{CT}(\mathtt{C})\ \mathrm{OK}}{\vdash \mathtt{CT}\ \mathrm{OK}} \quad (\text{T-PROGRAM})$$

Figure 3. IFJ: Typing rules for expressions, methods, classes and program CT w.r.t. the class signature table $\mathtt{CST} = \mathit{signature}(\mathtt{CT})$

In order to type the `null` value, we introduce the special type $\bot$, that is not a class name, cannot occur in IFJ programs and is a subtype of any other type. We will use the metavariable T to denote either a class name or $\bot$.

The IFJ typing rules are given in Figure 3. A type environment Γ is a mapping from variables (including `this`) to class names, written $\bar{\mathtt{x}} : \bar{\mathtt{C}}$. The empty environment will be denoted by $\bullet$. The rules for variable (T-VAR), field selection (T-FIELD), method invocation (T-INVK), object creation (T-NEW), upcast (T-UCAST), downcast (T-DCAST), method definition (T-METHOD) and class definition (T-CLASS) are analogous to the corresponding rules for FJ given in [16]. However, the presentation is slightly different since our rules refer to the class signature table of the program rather than to the class table. In particular, the rule for typing the definition of a method m in a class C, (T-METHOD), relies on the fact that, according to the sanity condition (*iv*) of the class signature table, any definition of a the method with name m in a superclass of C must have the same type. We also have a rule for `null` and a rule for field assignment (not contained in FJ) and a rule for typing the whole program (left implicit in FJ). Note that expressions like $(\mathtt{C})\mathtt{e}$ where the type of e is not a subtype of C (called *stupid casts* in [16]) or $\mathtt{null.f}$ and $\mathtt{null.m}(\cdots)$ (that we will call *stupid selections*) are ill-typed.

Following [16], we say that a well-typed IFJ program is *cast safe* to mean that it can be typed without using the rule for down-

DMD	::=	delta δ {$\overline{\texttt{DC}}$}	delta modules
DC	::=	adds CD | removes C |	delta clauses
		modifies C [extending C] { $\overline{\texttt{DS}}$ }	
DS	::=	adds FD | adds MD |	delta subclauses
		removes a | modifies MD	

Figure 4. IFΔJ: syntax of delta modules

cast. Every well-typed IFJ program is literally a well-typed JAVA program.

5. Imperative Featherweight Delta Java

In this section, we introduce the syntax and the semantics of IFΔJ, a core calculus for DOP of product lines of JAVA programs based on IFJ. A similar formalization of DOP is presented in [31] relying on LJ [33]. However, the goal in [31] is to formally show the embedding of FOP into DOP whereas in this work we focus on compositional type checking.

IFΔJ *Syntax* The abstract syntax of the IFΔJ constructs is given in Figure 4. The constructs for class definitions CD, field definitions FD and method definitions MD are those of IFJ, given in Figure 2. The metavariable δ ranges over delta module names.

A delta module definition DMD (see Figure 4) can be understood as a pair formed by the name δ of the delta module and by a mapping from class names to delta clause definitions. A delta clause definition DC can specify the addition, removal or modification of a class. The adds-domain, the removes-domain and the modifies-domain of a delta module definition DMD are defined as follows:

$$\begin{array}{lcl} \mathit{addsDom}(\texttt{DMD}) & = & \{\texttt{C} \mid \texttt{DMD(C)} = \texttt{adds class C} \cdots\} \\ \mathit{removesDom}(\texttt{DMD}) & = & \{\texttt{C} \mid \texttt{DMD(C)} = \texttt{removes C}\} \\ \mathit{modifiesDom}(\texttt{DMD}) & = & \{\texttt{C} \mid \texttt{DMD(C)} = \texttt{modifies C} \cdots\} \end{array}$$

The modification of a class is defined by possibly changing the super class and by listing a sequence of delta subclauses DS defining modifications of methods and additions/removals of fields and methods. A delta modifies-clause DC can be understood as a mapping from the keyword `extending` to an either empty or singleton set of class names and from field/method names to delta subclauses. The adds-, removes- and modifies-domain of a delta modifies-clause DC are defined as follows:

$$\begin{array}{lcl} \mathit{addsDom}(\texttt{DC}) & = & \{\texttt{a} \mid \texttt{DMT(a)} = \texttt{adds} \cdots \texttt{a} \cdots\} \\ \mathit{removesDom}(\texttt{DC}) & = & \{\texttt{a} \mid \texttt{DMT(a)} = \texttt{removes a}\} \\ \mathit{modifiesDom}(\texttt{DC}) & = & \{\texttt{m} \mid \texttt{DMT(m)} = \texttt{modifies} \cdots \texttt{m} \cdots\} \end{array}$$

The modification of a method, defined by a delta modifies-subclause, can either replace the method body by another implementation, or wrap the existing method using the `original` construct. In both cases, the modified method must have the same header as the unmodified method. The `original` construct is modeled by the special variable `original` that may only occur in the body of the method MD provided by a delta modifies-subclause `modifies` MD and, like the special variable `this`, may not be used as the name of a method's formal parameter. An occurrence of `original` represents a call to the unmodified method where the formal parameters of the modified method are passed implicitly as arguments. Therefore, the type assumed for `original` is the return type of the method. The `Super()` call of AHEAD is modeled in the same way in LFJ [12].

After we have defined the notion of delta modules over IFJ, we can formalize IFΔJ product lines. We use the metavariables φ and ψ to range over feature names. We write $\overline{\psi}$ as short for the set $\{\overline{\psi}\}$, i.e., the feature configuration containing the features $\overline{\psi}$. A *delta module table* DMT is a mapping from delta module names to delta module definitions. An IFΔJ SPL is a 5-tuple $\texttt{L} = (\overline{\varphi}, \Phi, \texttt{DMT}, \Gamma, \prec)$ consisting of:

1. the features $\overline{\varphi}$ of the SPL,
2. the set of the valid feature configurations $\Phi \subseteq \mathscr{P}(\overline{\varphi})$,[3]
3. a delta module table DMT containing the delta modules,
4. a mapping $\Gamma : \mathit{dom}(\texttt{DMT}) \to \Phi$ determining for which feature configurations a delta module must be applied (which is denoted by the `when` clause in the concrete examples),
5. a total order $\prec$ on a partition of $\mathit{dom}(\texttt{DMT})$, called the application order, determining the order of delta module application.

The 4-tuple $(\overline{\varphi}, \Phi, \Gamma, \prec)$ represents the *product line declaration*, while the delta module table DMT represents the *code base*. To simplify notation, in the following we always assume a *fixed* SPL $\texttt{L} = (\overline{\varphi}, \Phi, \texttt{DMT}, \Gamma, \prec)$.

We further assume that L satisfies the following sanity conditions.

(*i*) For every class name C (except `Object`) appearing in DMT, we have $\texttt{C} \in (\cup_{\delta \in \mathit{dom}(\texttt{DMT})} \mathit{addsDom}(\texttt{DMT}(\delta)))$, meaning that every class is added at least once.

(*ii*) The inverse of Γ, the mapping $\Gamma^{-1} : \Phi \to \mathscr{P}(\mathit{dom}(\texttt{DMT}))$, is injective and such that $(\cup_{\overline{\psi} \in \Phi} \Gamma^{-1}(\overline{\psi})) = \mathit{dom}(\texttt{DMT})$. I.e., for every feature configuration a different set of delta modules is applied and every delta module is applied for at least one feature configuration ($\Gamma^{-1}(\overline{\psi})$ is the set of names of delta modules whose application condition is satisfied by the feature configuration $\overline{\psi}$).

In the following, we write $\mathit{dom}(\delta)$ as short for $\mathit{dom}(\texttt{DMT}(\delta))$, and we write $\delta(\texttt{C})$ as short for $\texttt{DMT}(\delta)(\texttt{C})$.

IFΔJ *Product Generation* A delta module is *applicable* to a class table CT if each class to be removed or modified exists and, for every delta modifies clause, if each method or field to be removed exists, if each method to be modified exists and has the same header specified in method modifies subclause, and if each class, method or field to be added does not exist.

Given a delta module δ and a class table CT such that δ is applicable to CT, the application of δ to CT, denoted by $\textsc{apply}(\delta, \texttt{CT})$, is the class table CT′ defined as follows:

$$\texttt{CT}'(\texttt{C}) = \begin{cases} \texttt{CT(C)} & \text{if } \texttt{C} \notin \mathit{dom}(\texttt{DMT}(\delta)) \\ \texttt{CD} & \text{if } \delta(\texttt{C}) = \texttt{adds CD} \\ \textsc{apply}(\delta(\texttt{C}), \texttt{CT(C)}) & \text{if } \texttt{C} \in \mathit{modifiesDom}(\delta) \end{cases}$$

where $\textsc{apply}(\delta(\texttt{C}), \texttt{CT(C)})$, the application of the delta-modifies clause $\delta(\texttt{C}) = \texttt{DC}$ to the class definition $\texttt{CT(C)} = \texttt{CD}$, is the class definition CD′ defined as follows:

$$\texttt{CD}'(\texttt{extends}) = \begin{cases} \texttt{CD(extends)} & \text{if } \texttt{DC(extending)} = \emptyset \\ \texttt{C}' & \text{if } \texttt{DC(extending)} = \{\texttt{C}'\} \end{cases}$$

$$\texttt{CD}'(\texttt{a}) = \begin{cases} \texttt{CD(a)} & \text{if } \texttt{a} \notin \mathit{dom}(\texttt{DC}) \\ \texttt{AD} & \text{if } \texttt{DC(a)} = \texttt{adds AD} \\ \texttt{MD[e/original]} & \text{if } \texttt{DC(a)} = \texttt{modifies MD} \\ & \quad \text{and } \texttt{CD(a)} = \cdots\texttt{a}(\cdots)\{\texttt{return e;}\} \end{cases}$$

The semantics of the `original` construct is captured by replacing the occurrences of the special variable `original` in the method body specified by the modifies-subclause with the body of the unmodified method.

For any given total order of delta module application, an IFΔJ SPL defines a *product generation mapping*. That is, a partial map-

[3] The calculus abstracts from the concrete representation of the feature model.

ping from each feature configuration $\overline{\psi}$ in Φ to the class table of the product that is obtained by applying the delta modules $\Gamma^{-1}(\overline{\psi})$ to the empty class table according to the given order. The product generation mapping can be partial since a non-applicable delta module may be encountered during product generation such that the resulting product is undefined.

We write $\mathtt{CT}_{\overline{\psi}}$ to denote the class table generated for the feature configuration $\overline{\psi}$ and write $<:_{\overline{\psi}}$ and $aDef_{\overline{\psi}}$ to denote the subtype relation and the field/method lookup function associated with the class table $\mathtt{CT}_{\overline{\psi}}$, respectively.

Unambiguous, Local Unambiguous and Type-Safe* IFΔJ *Product Lines An IFΔJ SPL is *unambiguous* if all total orders of delta modules that are compatible with the application partial order define the same product generation mapping. In an unambiguous SPL, for every feature configuration at most one product implementation is generated.

In order to find a criterion for unambiguity, we define the notion of consistency of a set of delta modules. A set of delta modules is *consistent* if no class added or removed in one delta module is added, removed or modified in another delta module contained in the same set, and for every class modified in more than one delta module, its direct superclass is changed at most by one delta clause and the fields and methods added, modified or removed are distinct. For a consistent set of delta modules, any order of delta module application yields the same class table since the alterations do not interfere with each other. Consistency of a set of delta modules can be inferred by only considering delta module signatures that can be obtained by a straightforward inspection of each delta module in isolation. A *delta module signature* DMS is the analogue of a class signature for a delta module. The abstract syntax of delta module signatures is obtained from the syntax of delta modules, in Fig. 4, by replacing class definitions (CD) with class signatures (CS) and replacing method definitions (MD) with method headers (MH). We write $signature(\delta)$ to denote the signature of the delta module δ.

A SPL is *locally unambiguous* if every set S of delta modules in the partition of $dom(\mathtt{DMT})$ provided by the application partial order $\prec$ is consistent. If the SPL L is locally unambiguous, then it is also unambiguous. For local ambiguity, two delta modules that modify the same method cannot be placed in the same partition even if they are never applied together. However, local unambiguity can be checked by only relying on delta module signatures and the partition of $dom(\mathtt{DMT})$ provided by the application partial order. This makes local ambiguity robust to change since it is preserved by changes of delta modules without changing their signatures, by refinement of the application partial order, by changes to the application conditions and by changes in the set of valid feature configurations.

A IFΔJ SPL is *type-safe* if the following conditions hold: (i) its product generation mapping is total, (ii) it is locally unambiguous, and (iii) all generated products are well-typed IFJ programs.

6. Constraint-based Type System for IFJ

In this section, we present a constraint based type system for IFJ that is equivalent to the type system presented in Section 4. The constraint-based type system for IFJ infers a set of class constraints $\mathscr{F}$ for a given program CT. These constraints can then be checked against the class signature table $signature(\mathtt{CT})$ in order to establish whether CT is a well-typed IFJ program.

6.1 Flat Constraints and Flat Constraints Checking

In this section, we introduce some preliminary definitions that are used for the constraint-based IFJ type system.

Flat Constraints *Flat constraints*, illustrated in Figure 5, involve the type $\bot$, class names and type variables. Type variables, ranged over by α, β and γ, will be instantiated to class names when checking the constraints. The metavariable η denotes either a class name or a type variable, while the metavariable τ denotes either the type $\bot$, or a class name, or a type variable. We say that a constraint is *ground* to mean that it does not contain type variables.

$\mathbf{class}(\mathtt{C})$	class C must be defined
$\mathbf{subtype}(\tau,\eta)$	τ must be a subtype of η
$\mathbf{cast}(\mathtt{C},\tau)$	type τ must be castable to C
$\mathbf{field}(\eta,\mathtt{f},\alpha)$	class η must define or inherit field f of type α
$\mathbf{meth}(\eta,\mathtt{m},\overline{\alpha}\to\beta)$	class η must define or inherit method m of type $\overline{\alpha}\to\beta$

Figure 5. IFJ: Syntax and (informal) meaning of flat constraints

Checking Flat Constraints The checking judgment for flat constraints is $\mathtt{CST}\models\mathscr{F}$, to be read "the constraints in the set of flat constraints $\mathscr{F}$ are satisfied with respect to the class signature table CST". The associated rules are given in Figure 6 where $\uplus$ denotes the disjoint union of sets of constraints. The rules are almost self-explanatory, according to the informal meaning of the flat constraints given in Figure 5. The checking of a constraint of the form $\mathbf{subtype}(\cdots,\cdots)$ or $\mathbf{cast}(\cdots,\cdots)$ can be performed only when the constraint is ground. Note that there are two rules for checking a constraint of the form $\mathbf{cast}(\cdot,\cdot)$ corresponding to an upcast and to a downcast, respectively. The checking of a constraint of the form $\mathbf{field}(\cdot,\cdot,\cdot)$ or $\mathbf{meth}(\cdot,\cdot,\cdot)$ can be performed only when the first argument is a class name and the third argument contains types variables only. It causes the instantiation of all the type variables occurring in the third argument.

We say that a set of flat constraints is *cast safe with respect to a class signature table* CST to mean that it can be checked without using the rule associated with downcast.

6.2 Constraint-based Typing Rules for IFJ

The constraint-based typing rules for IFJ organize the inferred constraints in a two-level hierarchy, corresponding to the structure of the class table of the IFJ program. Namely: (i) the typing rules infer a set of *class constraints* (one for each class definition in the program); (ii) each class constraint consists of the name of the respective class C and of a set of *method constraints* inferred for the methods defined in the class; and (iii) each method constraint consists of the name of the respective method and of the set of flat constraints inferred for the body of the method. Thus, a set of class constraints $\mathscr{C}$ can be understood as a mapping from class names to class constraints, and a class constraint can be understood as a mapping from method names to method constraints. The syntax of class constraints and method constraints is given in Figure 7.

The constraint-based typing judgment for programs is $\vdash \mathtt{CT} : \mathscr{C}$, to be read "program CT has the class constraints $\mathscr{C}$". The constraint-based typing rules for IFJ expressions, methods, classes and programs are given in Figure 8. The rules (CT-FIELD) and (CT-INVK) are the only rules that create type variables. The type variables α created by rule (CT-FIELD) occur in the third argument of the flat constraint $\mathbf{field}(\eta,\mathtt{f},\alpha)$, and the type variables $\alpha_1,...,\alpha_n,\beta$ created by rule (CT-INVK) occur in the third argument of the flat constraint $\mathbf{meth}(\eta,\mathtt{m},\alpha_1\cdots\alpha_n\to\beta)$. Therefore, the checking rules for flat constraints (given in Section 6.1) can be applied by considering the constraints in the order in which they are created. In particular, when the constraints $\mathbf{field}(\cdot,\cdot,\cdot)$ and $\mathbf{meth}(\cdot,\cdot,\cdot)$ are checked their third arguments contain type variables only, and checking the constraints in a different order may not cause the instantiation of any

$$\mathtt{CST} \models \emptyset \qquad \frac{\mathtt{C} \in dom(\mathtt{CST}) \quad \mathtt{CST} \models \mathscr{F}}{\mathtt{CST} \models \{\mathbf{class}(\mathtt{C})\} \uplus \mathscr{F}} \qquad \frac{\mathtt{T} <: \mathtt{C}}{\mathtt{CST} \models \mathbf{subtype}(\mathtt{T},\mathtt{C})} \qquad \frac{\mathtt{T} <: \mathtt{C} \quad \mathtt{CST} \models \mathscr{F}}{\mathtt{CST} \models \{\mathbf{cast}(\mathtt{C},\mathtt{T})\} \uplus \mathscr{F}} \qquad \frac{\mathtt{C} <: \mathtt{T} \quad \mathtt{C} \neq \mathtt{T} \quad \mathtt{CST} \models \mathscr{F}}{\mathtt{CST} \models \{\mathbf{cast}(\mathtt{C},\mathtt{T})\} \uplus \mathscr{F}}$$

$$\frac{aType(\mathtt{C})(\mathtt{f}) = \mathtt{A} \quad \mathtt{CST} \models \mathscr{F}[\mathtt{A}/\alpha]}{\mathtt{CST} \models \{\mathbf{field}(\mathtt{C},\mathtt{f},\alpha)\} \uplus \mathscr{F}} \qquad \frac{aType(\mathtt{C})(\mathtt{m}) = \mathtt{A}_1 \cdots \mathtt{A}_n \to \mathtt{A}_0 \quad \mathtt{CST} \models \mathscr{F}[\mathtt{A}_0 \cdots \mathtt{A}_n/\alpha_0 \cdots \alpha_n]}{\mathtt{CST} \models \{\mathbf{meth}(\mathtt{C},\mathtt{m},\alpha_1 \cdots \alpha_n \to \alpha_0)\} \uplus \mathscr{F}}$$

Figure 6. IFJ: Checking rules for satisfaction of flat constraints w.r.t. a class signature table

of those type variables before that the corresponding constraint is checked.

Some expressions are recognized as ill-typed during constrain generation. Namely, method bodies containing: occurrences of variables `x` that are not declared as method parameters, according to rules (CT-METHOD) and (CT-VAR); or stupid selections (i.e., expressions like `null.f` and `null.m`$(\cdots)$), according to rules (CT-NULL), (CT-FIELD) and (CT-INVK).

The following example illustrates the constraint-based type system by considering an encoding in IFJ of methods from the EPL example introduced in Section 2.

EXAMPLE 6.1. The sequential composition of expressions, "$\mathtt{e}_1;\mathtt{e}_2;\mathtt{e}_3$", which is not part of the IFJ syntax, can be encoded as "`new Encode().sc3C(`$\mathtt{e}_1,\mathtt{e}_2,\mathtt{e}_3$`)`" where `C` is the type of $\mathtt{e}_3$ and the class `Encode` defines the method

```
C sc3C(Object x1, Object x2, C x3) { return x3; }
```

Therefore, the method `setAdd` of class `Add` introduced in Listing 1 can be encoded in IFJ as follows:

```
Add setAdd(Exp a, Exp b) {
 return new Encode().sc3Add(this.expr1=a, this.expr2=b, this); }
```

The inferred method constraint is

$$\mathtt{setAdd}\ \mathbf{with} \left\{ \begin{array}{l} \mathbf{field}(\mathtt{Add},\mathtt{expr1},\alpha'),\ \mathbf{subtype}(\mathtt{Exp},\alpha'), \\ \mathbf{field}(\mathtt{Add},\mathtt{expr2},\alpha''),\ \mathbf{subtype}(\mathtt{Exp},\alpha''), \\ \mathbf{subtype}(\mathtt{Exp},\mathtt{Object}),\ \mathbf{subtype}(\mathtt{Add},\mathtt{Add}),\ \mathbf{class}(\mathtt{Encode}), \\ \mathbf{meth}(\mathtt{Encode},\mathtt{sc3C},(\mathtt{Object},\mathtt{Object},\mathtt{Add} \to \mathtt{Add})) \end{array} \right\}$$

Some optimizations (not considered in this paper) are possible. Constraints like $\mathbf{subtype}(\mathtt{Exp},\mathtt{Object})$ and $\mathbf{subtype}(\mathtt{Add},\mathtt{Add})$ can be dropped. Information about standard library classes, like `Encode`, that cannot be modified by the delta modules can be exploited to infer simpler constraints. For example, the following simpler constraints could be inferred:

$$\mathtt{setAdd}\ \mathbf{with} \left\{ \begin{array}{l} \mathbf{field}(\mathtt{Add},\mathtt{expr1},\alpha'),\ \mathbf{subtype}(\mathtt{Exp},\alpha'), \\ \mathbf{field}(\mathtt{Add},\mathtt{expr2},\alpha''),\ \mathbf{subtype}(\mathtt{Exp},\alpha'') \end{array} \right\}$$

The method `eval` introduced in class `Add` by the delta module `DAddEval` in Listing 2 can be encoded in IFJ as follows:

```
Int eval() { return this.expr1.eval().sum(this.expr2.eval()); }
```

where `Int` is class for integers. The inferred class constraint is

$$\mathtt{eval}\ \mathbf{with} \left\{ \begin{array}{l} \mathbf{field}(\mathtt{Add},\mathtt{expr1},\beta'),\ \mathbf{meth}(\beta',\mathtt{eval},(\bullet \to \gamma')), \\ \mathbf{field}(\mathtt{Add},\mathtt{expr2},\beta''),\ \mathbf{meth}(\beta'',\mathtt{eval},(\bullet \to \gamma')), \\ \mathbf{meth}(\gamma',\mathtt{sum},(\gamma' \to \gamma'')),\ \mathbf{subtype}(\gamma'',\mathtt{Int}) \end{array} \right\}$$

Assuming that `Int` is a standard library final class, it would be possible to infer a simpler constraint (namely, replace γ'' by `Int` and drop $\mathbf{subtype}(\gamma'',\mathtt{Int})$). Further optimizations are possible in presence of primitive types (not formalized in IFJ). For instance, given the original version of method `eval` (that uses the primitive type `int`)

```
int eval() { return this.expr1.eval() + this.expr2.eval(); }
```

it would be possible to infer the simpler method constraint

$$\mathtt{eval}\ \mathbf{with} \left\{ \begin{array}{l} \mathbf{field}(\mathtt{Add},\mathtt{expr1},\beta'),\ \mathbf{meth}(\beta',\mathtt{eval},(\bullet \to \mathtt{int})), \\ \mathbf{field}(\mathtt{Add},\mathtt{expr2},\beta''),\ \mathbf{meth}(\beta'',\mathtt{eval},(\bullet \to \mathtt{int})) \end{array} \right\}$$

6.3 Properties of IFJ Constraint-based Typing

The hierarchical organization of the constraints derived for a product is immaterial for checking their satisfaction. The function FLAT transforms a set of class-constraints $\mathscr{C}$ into a set of flat constraints $\mathrm{FLAT}(\mathscr{C})$. It is defined as follows.

$$\begin{array}{rcl} \mathrm{FLAT}(\mathtt{C}_1\ \mathbf{with}\ \mathscr{K}_1,...,\mathtt{C}_n\ \mathbf{with}\ \mathscr{K}_n) & = & \cup_{i\in\{1,...,n\}}\mathrm{FLAT}(\mathscr{K}_i) \\ \mathrm{FLAT}(\{\mathtt{m}_1\ \mathbf{with}\ \mathscr{F}_1,...,\mathtt{m}_n\ \mathbf{with}\ \mathscr{F}_n\}) & = & \cup_{i\in\{1,...,n\}}\mathscr{F}_i \end{array}$$

The following theorem states that the constraint-based type system is correct and complete with respect to the IFJ type system given in Section 4.

THEOREM 6.2 (Correctness/Completeness). *Let* `CT` *be a* IFJ *program and* $\mathtt{CST} = signature(\mathtt{CT})$.

(Correctness) *Let* $\vdash \mathtt{CT} : \mathscr{C}$ *and* $\mathtt{CST} \models \mathrm{FLAT}(\mathscr{C})$. *Then*

1. $\vdash \mathtt{CT}$ OK, *and*
2. *if* $\mathrm{FLAT}(\mathscr{C})$ *is cast-safe with respect to* `CST`, *then* `CT` *is cast-safe.*

(Completeness) *Let* $\vdash \mathtt{CT}$ OK. *Then there exists* $\mathscr{C}$ *such that:*

1. $\vdash \mathtt{CT} : \mathscr{C}$ *and* $\mathtt{CST} \models \mathrm{FLAT}(\mathscr{C})$, *and*
2. *if* `CT` *is cast-safe, then* $\mathrm{FLAT}(\mathscr{C})$ *is cast-safe with respect to* `CST`.

7. Constraint-based Type System for IFΔJ

The constraint-based type system for IFΔJ analyzes each delta module in isolation. The results of the analysis can be combined with the product line declaration in order to check whether all the products that can be generated are well-typed.

For each product configuration $\overline{\psi}$, the class signature table $\mathtt{CST}_{\overline{\psi}}$ of the product $\mathtt{CT}_{\overline{\psi}}$ can be generated by applying the signature of the delta modules in $\Gamma^{-1}(\overline{\psi})$ to the empty class signature table according to the given order (similarly to product generation). The constraint-based type system infers, for each delta module, a set of delta clause constraints $\mathscr{D}_\delta$. The inferred sets of delta clause constraints are organized such that, for each product configuration $\overline{\psi}$, the set of class constraints $\mathscr{C}_{\overline{\psi}}$ of the product $\mathtt{CT}_{\overline{\psi}}$ can be derived by applying the sets of delta clause constraints inferred for the delta modules in $\Gamma^{-1}(\overline{\psi})$ to the empty set of class constraints. Therefore, the type safety of a product line can be established by relying only on the the delta module signatures, the delta clause-constraints and the product line declaration without re-inspecting the delta modules and without generating the products.

7.1 Constraint-based Typing Rules for Delta Modules

The typing rules for delta modules also organize the inferred delta clause constraints in a two-level hierarchy corresponding to the structure of the delta module to support the application of a set of delta clause constraints $\mathscr{D}$ to a set of class constraints $\mathscr{C}$. The syntax of the delta clause constraints is given in Figure 9. The typing rules infer a set of *delta adds/removes/modifies-clause constraints* (one for each delta clause in the delta module). A delta adds-clause constraint consists of the keyword `adds` followed by a class constraint (defined in Figure 7). A delta removes-clause constraint is a removes-clause `removes C`. Each delta modifies-clause

Method constraints:

$\mathtt{m}$ **with** $\mathscr{F}$ method $\mathtt{m}$ has the set of flat constraints $\mathscr{F}$

Class constraints:

$\mathtt{C}$ **with** $\mathscr{K}$ class $\mathtt{C}$ has the set of method constraints $\mathscr{K}$

Figure 7. IFΔJ: Syntax of class constraints

Expression typing:

$$\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \mid \emptyset \quad (\text{CT-VAR})$$

$$\frac{\Gamma \vdash \mathtt{e} : \eta \mid \mathscr{F} \qquad \alpha \text{ fresh}}{\Gamma \vdash \mathtt{e.f} : \alpha \mid \{\mathbf{field}(\eta, \mathtt{f}, \alpha)\} \cup \mathscr{F}} \quad (\text{CT-FIELD})$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \eta \mid \mathscr{F}_0 \qquad \alpha_1, ..., \alpha_n, \beta \text{ fresh} \qquad \forall i \in 1..n, \quad \Gamma \vdash \mathtt{e}_i : \tau_i \mid \mathscr{F}_i}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\mathtt{e}_1, ..., \mathtt{e}_n) : \beta \mid \{\mathbf{meth}(\eta, \mathtt{m}, \alpha_1 \cdots \alpha_n \to \beta), \mathbf{subtype}(\tau_1, \alpha_1), \ldots, \mathbf{subtype}(\tau_n, \alpha_n)\} \cup (\cup_{i \in \{0,...,n\}} \mathscr{F}_i)} \quad (\text{CT-INVK})$$

$$\frac{\mathtt{C} \in dom(\mathtt{ACST}_\mathtt{L})}{\Gamma \vdash \mathtt{new\ C}() : \mathtt{C} \mid \{\mathbf{class}(\mathtt{C})\}} \quad (\text{CT-NEW}) \qquad \frac{\Gamma \vdash \mathtt{e} : \tau \mid \mathscr{F}}{\Gamma \vdash (\mathtt{C})\mathtt{e} : \mathtt{C} \mid \{\mathbf{cast}(\mathtt{C}, \tau)\} \cup \mathscr{F}} \quad (\text{CT-CAST})$$

$$\Gamma \vdash \mathtt{null} : \bot \mid \emptyset \quad (\text{CT-NULL}) \qquad \frac{\Gamma \vdash \mathtt{e}_0.\mathtt{f} : \eta \mid \mathscr{F}_0 \qquad \Gamma \vdash \mathtt{e}_1 : \tau \mid \mathscr{F}_1}{\Gamma \vdash \mathtt{e}_0.\mathtt{f} = \mathtt{e}_1 : \eta \mid \{\mathbf{subtype}(\tau, \eta)\} \cup \mathscr{F}_0 \cup \mathscr{F}_1} \quad (\text{CT-ASSIG})$$

Method definition typing:

$$\frac{\mathtt{this} : \mathtt{C}, \mathtt{original} : \mathtt{B}, \bar{\mathtt{x}} : \bar{\mathtt{A}} \vdash \mathtt{e} : \tau \mid \mathscr{F}}{\mathtt{this} : \mathtt{C} \vdash \mathtt{B\ m}\ (\bar{\mathtt{A}}\ \bar{\mathtt{x}})\{\mathtt{return\ e;}\} : \mathtt{m}\ \mathbf{with}\ (\{\mathbf{subtype}(\tau, \mathtt{B})\} \cup \mathscr{F})} \quad (\text{CT-METHOD})$$

Class definition typing:

$$\frac{\forall i \in 1..q, \quad \mathtt{this} : \mathtt{C} \vdash \mathtt{MD}_i : \{\mathtt{m}_i\ \mathbf{with}\ \mathscr{F}_i\}}{\vdash \mathtt{class\ C\ extends\ D}\ \{\ \overline{\mathtt{FD}};\ \mathtt{MD}_1 \cdots \mathtt{MD}_q\ \} : \mathtt{C}\ \mathbf{with}\ \mathtt{m}_1\ \mathbf{with}\ \mathscr{F}_1, ..., \mathtt{m}_q\ \mathbf{with}\ \mathscr{F}_q} \quad (\text{CT-CLASS})$$

Program typing:

$$\frac{dom(\mathtt{CT}) = \{\mathtt{C}_1, ..., \mathtt{C}_n\} (n \geq 0) \qquad \forall i \in 1..n, \quad \vdash \mathtt{CT}(\mathtt{C}_i) : \mathtt{C}_i\ \mathbf{with}\ \mathscr{K}_i}{\vdash \mathtt{CT} : \{\mathtt{C}_1\ \mathbf{with}\ \mathscr{K}_1, ..., \mathtt{C}_n\ \mathbf{with}\ \mathscr{K}_n\}} \quad (\text{CT-PROGRAM})$$

Figure 8. IFJ: Constraint-based typing rules for expressions, methods, classes and programs

constraint consists of the name of the subject class C and of a set of *delta adds/removes/replaces/wraps-subclause constraints* that are described as follows:

- An adds-subclause constraint consists of the keyword `adds` followed by a method constraint (defined in Figure 7).
- A delta removes-subclause constraint is of the form `removes m`.
- A wraps/replaces-clause constraint consists of the keyword `replaces/wraps` followed by a method constraint (defined in Figure 7). Wrap-subclause constraints are inferred for modify-subclauses containing `original`, while replace-subclause constraints are inferred for modify-subclauses not containing `original`.

Thus, a set of delta clause constraints $\mathscr{D}$ can be understood as a mapping from class names to delta clause constraints, and a delta modifies-clause constraint can be understood as a mapping from method names to delta subclause constraints.

The constraint-based typing judgment for a delta module is $\vdash \mathtt{delta}\ \delta\ \cdots : \mathscr{D}_\delta$, to be read as "the delta module δ has the delta clause constraints $\mathscr{D}_\delta$". The constraint-based typing rules for IFΔJ delta subclauses, delta clauses and delta modules are given in Figure 10. Most of the rules are self-explanatory, according to the meaning of delta clause constraints and delta subclause constraints illustrated above. The rules for the delta subclauses that add and remove a field ((CT-S-ADDF) and (CT-S-REMF), respectively) generate the empty set of constraints, since the checks associated to field declarations in a product are encompassed by the sanity conditions of the class signature table of the product. The rules (CT-S-ADDM), (CT-S-REPM), (CT-S-WRAM) and (CT-C-ADDC) rely on the rules (CT-METHOD) and (CT-CLASS) in Figure 8. The rule (CT-C-MODC) has an optional part (enclosed in square brackets) to cope with the fact that the `extending` part of a delta modifies-clause is optional.

7.2 Class Signature Tables and Class Constraints

The application of a delta module signature to a class signature table, denoted by APPLY(DMS, CST), performs the alterations specified in DMS to CST. We do not present the formal definition of APPLY(DMS, CST) since it mimics the application of a delta module to a class table presented in Section 5. The following proposition states that, given a delta module δ and a class table CT, the signature of the class table APPLY(δ, CT) can be computed directly from *signature*(δ) and *signature*(CT).

PROPOSITION 7.1. *If the delta module δ is applicable to the class table* CT*, then* APPLY(*signature*(δ), *signature*(CT)) = *signature*(APPLY(δ, CT)).

Therefore, for each feature configuration $\overline{\psi}$ in Φ, the class signature table of the corresponding product can be computed by applying the signatures of the delta modules $\Gamma^{-1}(\overline{\psi})$ to the empty class signature table according to the `after` partial order. We write $\mathtt{CST}_{\overline{\psi}}$ to denote the class signature table of the product for feature configuration $\overline{\psi}$.

The result of the application of a set of delta clause constraints $\mathscr{D}$ to a set of class constraints $\mathscr{C}$, denoted by APPLY($\mathscr{D}, \mathscr{C}$), is the

Delta-clause constraints:

`adds C with` $\mathscr{K}$	add the constraint "`C with` $\mathscr{K}$"
`removes C`	remove constraint "`C with` $\cdots$"
`modifies C with` $\mathscr{M}$	change the constraint "`C with` $\mathscr{K}$" into "APPLY(`modifies C with` $\mathscr{M}$, `C with` $\mathscr{K}$)"

Delta subclause-constraints:

`adds m with` $\mathscr{F}$	add the constraint "`m with` $\cdots$"
`removes m`	remove constraint "`m with` $\cdots$"
`replaces m with` $\mathscr{F}'$	change constraint "`m with` $\mathscr{F}$" into "`m with` $\mathscr{F}'$"
`wraps m with` $\mathscr{F}'$	change constraint "`m with` $\mathscr{F}$" into "`m with` $\mathscr{F} \cup \mathscr{F}'$"

Figure 9. IFΔJ: Syntax of delta clause constraints

Delta-subclause typing:

$$\texttt{this} : \texttt{C} \vdash \texttt{adds Df} : \emptyset \quad \text{(CT-S-ADDF)}$$

$$\frac{\texttt{this} : \texttt{C} \vdash \texttt{MD} : \texttt{m with } \mathscr{F} \quad \texttt{original} \notin \texttt{MD}}{\texttt{this} : \texttt{C} \vdash \texttt{adds MD} : \{\texttt{adds m with } \mathscr{F}\}} \quad \text{(CT-S-ADDM)}$$

$$\texttt{this} : \texttt{C} \vdash \texttt{removes f} : \emptyset \quad \text{(CT-S-REMF)}$$

$$\texttt{this} : \texttt{C} \vdash \texttt{removes m} : \{\texttt{removes m}\} \quad \text{(CT-S-REMM)}$$

$$\frac{\texttt{this} : \texttt{C} \vdash \texttt{MD} : \texttt{m with } \mathscr{F} \quad \texttt{original} \notin \texttt{MD}}{\texttt{this} : \texttt{C} \vdash \texttt{modifies MD} : \{\texttt{replaces m with } \mathscr{F}\}} \quad \text{(CT-S-REPM)}$$

$$\frac{\texttt{this} : \texttt{C} \vdash \texttt{MD} : \texttt{m with } \mathscr{F} \quad \texttt{original} \in \texttt{MD}}{\texttt{this} : \texttt{C} \vdash \texttt{modifies MD} : \{\texttt{wraps m with } \mathscr{F}\}} \quad \text{(CT-S-WRAM)}$$

Delta-clause typing:

$$\frac{\vdash \texttt{CD} : \texttt{C with } \mathscr{K}}{\vdash \texttt{adds CD} : \texttt{adds C with } \mathscr{K}} \quad \text{(CT-C-ADDC)}$$

$$\vdash \texttt{removes C} : \texttt{removes C} \quad \text{(CT-C-REMC)}$$

$$\frac{\forall i \in 1..q, \quad \texttt{this} : \texttt{C} \vdash \texttt{DS}_i : \mathscr{S}_i}{\vdash \texttt{modifies C [extending D] } \{\ \texttt{DS}_1 \ldots \texttt{DS}_q\ \} : \texttt{modifies C with } (\cup_{i \in \{1,\ldots,q\}} \mathscr{S}_i)} \quad \text{(CT-C-MODC)}$$

Delta-module typing:

$$\frac{\forall i \in 1..n, \quad \vdash \texttt{DC}_i : dcc_i}{\vdash \texttt{delta } \delta\ \{\texttt{DC}_1 \ldots \texttt{DC}_n\} : \{dcc_1, \ldots, dcc_n\}} \quad \text{(CT-DELTA)}$$

Figure 10. IFΔJ: Constraint-based typing rules for delta subclauses, delta clauses and delta modules

set of class constraints $\mathscr{C}'$ defined as follows:

$$\mathscr{C}'(\texttt{C}) = \begin{cases} \mathscr{C}(\texttt{C}) & \text{if } \texttt{C} \notin dom(\mathscr{D}) \\ \texttt{C with } \mathscr{K} & \text{if } \texttt{C} \notin dom(\mathscr{C}) \\ & \quad \text{and } \texttt{adds C with } \mathscr{K} \in \mathscr{D} \\ \text{APPLY}(\mathscr{D}(\texttt{C}), \mathscr{C}(\texttt{C})) & \text{if } \texttt{modifies C} \cdots \in \mathscr{D} \end{cases}$$

where the application of the delta modifies-clause constraint $dcc = \texttt{modifies C with } \mathscr{M} = \mathscr{D}(\texttt{C})$ to the class-constraint $cc = \texttt{C with } \mathscr{K} = \mathscr{C}(\texttt{C})$, denoted by $\text{APPLY}(dcc, cc)$, is the class-constraint $cc' = \texttt{C with } \mathscr{K}'$ defined as follows:

$$cc'(\texttt{m}) = \begin{cases} cc(\texttt{m}) & \text{if } \texttt{removes m} \cdots \notin dcc \\ & \quad \text{and } \texttt{modifies m} \cdots \notin dcc \\ \texttt{m with } \mathscr{F} & \text{if } dcc(\texttt{m}) = \texttt{adds m with } \mathscr{F} \\ & \quad \text{or } dcc(\texttt{m}) = \texttt{replaces m with } \mathscr{F} \\ \texttt{m with } \mathscr{F} \cup \mathscr{F}' & \text{if } dcc(\texttt{m}) = \texttt{wraps m with } \mathscr{F}' \\ & \quad \text{and } cc(\texttt{m}) = \texttt{m with } \mathscr{F} \end{cases}$$

The following proposition states that the constraint application operation defined above indeed allows computing the class constraints for the class table $\text{APPLY}(\delta, \texttt{CT})$ directly from the delta clause-constraints for δ and the class constraints for CT.

PROPOSITION 7.2. *For every delta module* $\delta \in dom(\texttt{DMT})$ *and for every class table* CT *such that* δ *is applicable to* CT*, if* $\vdash \texttt{DMT}(\delta) : \mathscr{D}_\delta$ *and* $\vdash \texttt{CT} : \mathscr{C}$*, then* $\vdash \text{APPLY}(\delta, \texttt{CT}) : \text{APPLY}(\mathscr{D}_\delta, \mathscr{C})$.

Therefore, for each feature configuration $\overline{\psi}$ in Φ, the class constraints $\mathscr{C}_{\overline{\psi}}$ for the product $\texttt{CT}_{\overline{\psi}}$ can be generated (without generating the product) by applying the sets of delta clause constraints inferred for the delta modules $\Gamma^{-1}(\overline{\psi})$ to the empty set of class constraints according to the `after` partial order.

7.3 Properties of IFΔJ Constraint-based Typing

The IFΔJ constraint-based type system enables checking the well-typedness of all possible products by analyzing the delta modules in isolation, generating the constraints for the products, and checking the constraints obtained for each product against the class signature table of that product. The following theorem (together with Theorem 6.2) states that the IFΔJ constraint-based typed system is correct and complete with respect to the IFJ type system.

THEOREM 7.3 (Correctness/Completeness of IFΔJ typing). *Let* L *be a locally unambiguous* IFΔJ *SPL and* $\overline{\psi} \in \Phi$.

(Correctness) *Let* $\vdash \texttt{delta } \delta \cdots : \mathscr{D}_\delta$ *for all* $\delta \in \Gamma^{-1}(\overline{\psi})$ *and* $\texttt{CST}_{\overline{\psi}} \models \text{FLAT}(\mathscr{C}_{\overline{\psi}})$. *Then:*

1. $\vdash \texttt{CT}_{\overline{\psi}}$ OK, *and*
2. *if* $\text{FLAT}(\mathscr{C}_{\overline{\psi}})$ *is cast-safe with respect to* $\texttt{CST}_{\overline{\psi}}$*, then* $\texttt{CT}_{\overline{\psi}}$ *is cast-safe.*

(Completeness) *Let* $\vdash \texttt{CT}_{\overline{\psi}}$ OK.

1. *If for all* $\delta \in \Gamma^{-1}(\overline{\psi})$ *there exists* $\mathscr{D}_\delta$ *such that* $\vdash \texttt{delta } \delta \cdots : \mathscr{D}_\delta$*, then*
   (a) $\texttt{CST}_{\overline{\psi}} \models \text{FLAT}(\mathscr{C}_{\overline{\psi}})$, *and*
   (b) *if* $\texttt{CT}_{\overline{\psi}}$ *is cast-safe then* $\text{FLAT}(\mathscr{C}_{\overline{\psi}})$ *is cast-safe with respect to* $\texttt{CST}_{\overline{\psi}}$.
2. *If there exists* $\delta \in \Gamma^{-1}(\overline{\psi})$ *such that* δ *is not* $\vdash$*-typable, then the body of the method adds/modifies-subclauses in* δ *that is ill typed is not included in the product* $\texttt{CT}_{\overline{\psi}}$.

8. Comparing DOP and AOP

Both delta-oriented and aspect-oriented programming (AOP) [23] combine code taken from different sources. In AOP, cross-cutting features (such as logging services or concurrency primitives) are factored out into aspects instead of scattering them in the application code. In DOP, deltas modules are the building blocks used to generate code implementing desired product features. Aspects refer to parts of a program at *join-points*, specified by *point-cut* expressions. By *advice*, the execution of the code at join-points can be modified. Advice can be defined to be executed *after*, *before* or *around* the "intercepted" join-point. In particular, an around-advice replaces the original code. The intercepted join-point can be executed using `proceed`, which corresponds to the `original` construct in DOP. The join-points can be of different nature, starting from the invocation of a specific method on an object of a specific class, to control-flow based execution points.

ASPECTJ [22], an extension of JAVA with aspects, provides a compiler generating standard JAVA code by applying aspects to JAVA classes. This process (*aspect weaving*) ensures that aspect and non-aspect code run together in the expected way. Aspect weaving, in ASPECTJ, is mostly carried out at compile time, reducing runtime overhead. For instance, most of the code inserted to intercept join-points by execution of advice is realized by an additional method invocation. Since such an invocation is typically a static or final method, it can be inlined by most JVMs. This means that the detection of most join-points according to the specified point-cuts can be performed statically by the ASPECTJ compiler. However, join-points can also have a dynamic nature, e.g., based on the dynamic type of objects referred to in the point-cuts, or based on the control-flow of the program, such as the first execution instance of a recursive method. Hence, aspects allow the programmer to intercept most execution statements in a program, also based on the dynamic control flow or the run-time type of objects.

The modification operations that can be specified in delta modules are sufficient to express before, after and around advice considered in AOP. Additionally, delta modules can change the superclass, change method implementations, and even remove methods, etc, while aspects cannot change types in a program statically. The partial ordering of delta modules provided by DOP product line declarations resembles the precedence order on advice in AOP. Delta modules do not comprise a specification formalism for modifications to be carried out at several places of a program. Instead, delta modules are statically connected to the product features, since delta application is performed at compile-time only. However, adding a flexible point-cut specification technique to delta modules is an interesting issue and a subject of future work.

A crucial difference between AOP and DOP is that the combination of aspects with a base program usually defines a single aspect-oriented program. An exception is the case when there are multiple ASPECTJ aspects without defined precedence. Then, the ASPECTJ compiler non-deterministically chooses one program to compile, such that a set of possible "woven" programs is defined. On the contrary, in DOP, a set of delta modules and a product line declaration defines a set of products.

The *A* calculus [14] aims at providing a core language as foundation of AOP. It solves the problem that type soundness in the presence of some around advice definitions breaks which also exists in the ASPECTJ implementation. The solution uses a flexible notion of `proceed`, by representing it with a simple term variable that denotes a closure. The notion of `proceed` in the *A* calculus is more flexible than other formalizations (see, e.g., [11, 15]), by relying on *type ranges* (while guaranteeing type soundness). In DOP, the `original` construct is similar to `proceed`. However, the modification of a method by a delta module does not change the signature, such that `original` can be used if a wrapper method with the same name should be defined. The `original` construct is, thus, typed with the same type as the return type of the original method (cf. Section 5).

9. Related Work

Delta-oriented programming [29, 31] is an extension of feature-oriented programming [6]. In [29], the general ideas of DOP are presented and compared conceptually and empirically to FOP. The presentation in [29] uses the notion of a core product, that is a designated product of the product line and the starting point of product generation. In [31], the notion of the core product is dropped such that product generation only relies on delta modules. This makes DOP even more flexible to support reactive, proactive and extractive product line engineering [24] and allows a direct embedding of FOP into DOP.

DOP and FOP are compositional approaches [20] for implementing SPLs in which code fragments are associated to product features and assembled to implement a particular feature configuration. Other compositional approaches use, for instance, aspects [4, 19], mixins [32], hyperslices [34] or traits [8, 13] to implement product line variability.

In this paper, we use the notion of DOP presented in [31] in order to provide a compositional approach for type checking delta-oriented product lines of JAVA programs. Various approaches to ensure the type safety of feature-oriented product lines can be found in literature [2, 3, 12, 25, 35]. The relations with the type system of LFJ [12], which is the closest to our proposal, has been already discussed in Section 3. The LFJ and the IFΔJ type systems have similarities with a type system proposed in [1] to type-check, compile and link code fragments. These code fragments, like feature/delta modules, can reference definitions provided in other code fragments. However, the purpose of the type system presented in [1] is to endure that linking code fragments compiled in isolation produces the same bytecode as the one that would be generated by the global compilation process performed by a standard JAVA compiler.

The FEATHERWEIGHT FEATURE JAVA for Product Lines (FFJ_{PL}) calculus [2] proposes an independently developed type checking approach for feature-oriented product lines. FFJ_{PL} relies on FFJ [3], a calculus for stepwise-refinement, that is not explicitly bound to implementing SPLs. FFJ is based on FJ (FEATHERWEIGHT JAVA) [16]. The main differences between FFJ_{PL} and LFJ are the following: (i) In FFJ_{PL}, feature-oriented mechanisms, such as class/method refinements, are modeled directly by the dynamic semantics of the language instead of by a translation into JAVA code; (ii) The FFJ_{PL} typing rules do not generate constraints, but directly consult the feature model, thus making it possible to straightforwardly identify the location of an error in the code; and (iii) FFJ_{PL} does not support modular type-checking (i.e., it does not meet the second requirement described in Section 3), since each feature module is analyzed by relying on information of the complete product line.

The design of the constraint-based type system for DOP involves issues similar to those considered in type-checking of dynamic classes [17]. Dynamic classes perform run-time updates of object-oriented systems by adding or refining classes (in a type-safe manner) or by removing redundant program parts. Each dynamic class is statically typed with a set of constraints (similar to the ones used for DOP) which are evaluated at run-time to ensure that the system is still well-typed after the update is carried out. Since dynamic classes are applied at run-time, only removals of redundant information are permitted in order to locally check applicability at run-time. DOP is a generative programming approach where variability is resolved at compile-time, such that DOP allows more flexible removals.

10. Conclusions and Future Work

We provided a foundation for compositional type checking of delta-oriented product lines that allows analyzing delta modules in isolation such that the analysis results can be re-used across different product lines. We illustrated our proposal considering the minimal core calculus IFJ [7], an imperative variant of FJ [16] (dealing with interfaces and exceptions, which are not included in the calculus, should not pose major technical problems). Two interesting and very challenging directions for future work are to extend the proposed type checking mechanism for DOP with the third step of the type checking approach pursued for LFJ (see Section 3) and to equip IFΔJ with a direct semantics as done for FFJ_{PL} [2] in order to foster the change of the feature configuration of a product at run-time [27, 28].

The concept of DOP is not bound to a particular programming language. For future work, we are aiming to consider other languages for the underlying product implementations. A starting point is the trait-based calculus FEATHERWEIGHT RECORD-TRAIT JAVA (FRTJ) [8, 9]. In FRTJ, classes are assembled from interfaces, records (providing fields) and traits [13] (providing methods) that can be directly manipulated by designated composition operations. These operations make FRTJ a good candidate for enhancing the flexibility of delta modules and providing further support for code reuse.

Acknowledgments. We are grateful to the anonymous AOSD 2011 referees and to the referees of earlier versions of this paper for many insightful comments and suggestions. We also thank Denis Meglio and Fabio Strocco for developing a prototypical implementation of a tool based on preliminary version of the IFΔJ calculus.

References

[1] D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Polymorphic Bytecode: Compositional Compilation for Java-like Languages. In *Proc. of POPL*, pages 26–37. ACM, 2005.

[2] S. Apel, C. Kästner, A. Grösslinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17(3):251–300, 2010.

[3] S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *GPCE*, pages 101–112. ACM, 2008.

[4] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. *Transactions on Aspect-Oriented Software Development*, 3880:135–173, 2006.

[5] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

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

[7] L. Bettini, F. Damiani, and I. Schaefer. IFJ: a Minimal Imperative Variant of FJ. Technical Report 133/2010, Dipartimento di Informatica, Università di Torino, 2010. Available from `http://www.di.unito.it/~damiani/papers/tr-133-2010.pdf`.

[8] L. Bettini, F. Damiani, and I. Schaefer. Implementing Software Product Lines using Traits. In *SAC, OOPS Track*, pages 2096–2102. ACM, 2010.

[9] L. Bettini, F. Damiani, I. Schaefer, and F. Strocco. A Prototypical Java-like Language with Records and Traits. In *PPPJ*, pages 129–138. ACM, 2010.

[10] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Longman, 2001.

[11] C. Clifton and G. T. Leavens. $MiniMAO_1$: Investigating the Semantics of Proceed. *SCP*, 63(3):321–374, Dec. 2006.

[12] B. Delaware, W. Cook, and D. Batory. A Machine-Checked Model of Safe Composition. In *FOAL*, pages 31–35. ACM, 2009.

[13] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2):331–388, 2006.

[14] B. D. Fraine, E. Ernst, and M. Südholt. Essential AOP: The A Calculus. In *ECOOP*, volume 6183 of *LNCS*, pages 101–125. Springer, 2010.

[15] B. D. Fraine, M. Südholt, and V. Jonckers. StrongAspectJ: flexible and safe pointcut/advice bindings. In *AOSD*, pages 60–71. ACM, 2008.

[16] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

[17] E. B. Johnsen, M. Kyas, and I. C. Yu. Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In *FM*, volume 5850 of *LNCS*, pages 596–611. Springer, 2009.

[18] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie Mellon Software Engineering Institute, 1990.

[19] C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features Using AspectJ. In *SPLC*, pages 223–232. IEEE, 2007.

[20] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *ICSE*, pages 311–320. ACM, 2008.

[21] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *SPLC*, pages 181–190. ACM, 2009.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[23] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

[24] C. Krueger. Eliminating the Adoption Barrier. *IEEE Software*, 19(4):29–31, 2002.

[25] M. Kuhlemann, D. Batory, and C. Kästner. Safe composition of non-monotonic features. In *GPCE*, pages 177–186. ACM, 2009.

[26] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *ECOOP*, volume 3586 of *LNCS*, pages 169–194. Springer, 2005.

[27] K. Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP*, volume 2374 of *LNCS*, pages 89–110. Springer, 2002.

[28] M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. Code generation to support static and dynamic composition of software product lines. In *GPCE*, pages 3–12. ACM, 2008.

[29] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented Programming of Software Product Lines. In *SPLC*, volume 6287 of *LNCS*, pages 77–91. Springer, 2010.

[30] I. Schaefer, L. Bettini, and F. Damiani. Compositional Type-Checking for Delta-Oriented Programming (version with Appendix). Technical Report 134/2010, Dipartimento di Informatica, Università di Torino, 2010. Available from `http://www.di.unito.it/~damiani/papers/tr-134-2010.pdf`.

[31] I. Schaefer and F. Damiani. Pure Delta-oriented Programming. In *FOSD 2010*, 2010. Available from `http://www.fosd.de/2010`.

[32] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[33] R. Strniša, P. Sewell, and M. Parkinson. The Java module system: core design and semantic definition. In *OOPSLA*, pages 499–514. ACM, 2007.

[34] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *ICSE*, pages 107–119, 1999.

[35] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *GPCE*, pages 95–104. ACM, 2007.

[36] M. Torgersen. The Expression Problem Revisited. In *ECOOP*, volume 3086 of *LNCS*, pages 123–146. Springer, 2004.