Acta Informatica (2013) 50:77–122
DOI 10.1007/s00236-012-0173-z

ORIGINAL ARTICLE

Compositional type checking of delta-oriented software product lines

Lorenzo Bettini · Ferruccio Damiani · Ina Schaefer

Received: 16 November 2011 / Accepted: 13 November 2012 / Published online: 11 January 2013

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. A particular product in a delta-oriented product line is generated by applying the modifications contained in the suitable delta modules to the empty program. 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 the results of the analysis can be reused. By relying only on the analysis results for the delta modules and on the product line declaration, it is possible to establish whether all the products of the product line are well typed according to the fragment of the JAVA type system modeled by the calculus.

The authors of this paper are listed in alphabetical order. This work has been partially supported by the Deutsche Forschungsgemeinschaft (DFG), the Italian MIUR project PRIN 2008 DISCO, the German-Italian University Centre (Vigoni program) and the EU project FP7-231620 HATS.

L. Bettini · F. Damiani
Dipartimento di Informatica, Universitá di Torino,
C.so Svizzera, 185, 10149 Torino, Italy
e-mail: lorenzo.bettini@unito.it

F. Damiani
e-mail: ferruccio.damiani@unito.it

I. Schaefer (✉)
Technische Universität Braunschweig, Mühlenpfordtstr. 23,
38106 Braunschweig, Germany
e-mail: i.schaefer@tu-bs.de; i.schaefer@tu-braunschweig.de

1 Introduction

Delta-oriented programming (DOP) [42,44] is a flexible compositional approach for implementing software product lines [12]. 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 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. A particular product in a delta-oriented product line is generated by applying the modifications contained in the suitable delta modules to the empty program. The product line declarationprovides the connection between the delta modules and the variabilities of the products defined in terms of product features and describes the set of valid feature configurations [28]. 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 reusing the same delta modules.

Delta-oriented programming is an extension of *feature-oriented programming* (FOP) [6], a compositional approach for implementing software product lines (cf. [44] 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 [34] which starts from existing product implementations. Moreover, the application conditions associated with delta modules, in the product line declaration, allow handling combinations of features explicitly. This provides an elegant way to counter the optional-feature problem [31] where two optional features require additional glue code to cooperate properly. However, the additional flexibility provided by DOP makes it more challenging than in FOP to efficiently check that for every valid feature configuration a unique product can be generated and that all the products of the product line are well typed.

Product line analysis techniques can be classified into three main categories [49]: *Product-based analyses* consider each product variant separately; *Family-based analyses* check the complete code base of the product line in a single run to obtain a result about all possible variants; *Feature-based analyses* consider the building blocks of the different product variants (the feature modules in FOP and the delta modules in DOP) in isolation to derive results on all variants.

In this paper, we provide a foundation for compositional type checking of delta-oriented product lines by presenting a constraint-based type system that supports a feature-based analysis phase and a final product-based analysis phase by relying on an abstraction of product generation. The concepts of this approach are demonstrated for IFΔJ (Imperative Featherweight Delta Java), a core calculus for delta-oriented product lines of Java programs. IFΔJ is based on IFJ (Imperative Featherweight Java), an imperative variant of FJ (Featherweight Java) [26], that is used to implement the products. In the feature-based analysis phase the constraint-based type system is used to analyze each delta module in isolation. The analysis result for a delta module is called the *type abstraction* of the delta module. In the product-based analysis phase, by relying only on the type abstractions of the delta modules and on the product line declaration, it is possible to

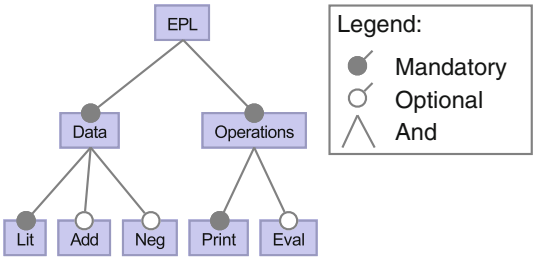

Fig. 1 Feature model for the expression product line

establish whether all the products of the product line are well typed according to the IFJ type system. The type abstraction of a delta module represents the provides/requires interface of the module. A novelty with respect to provide/requires interfaces of feature modules used by compositional type checking for FOP product lines [19] is that (during the final product-based analysis phase) the type abstraction of delta modules are composed in a delta-oriented manner.

The paper is organized as follows. Section 2 introduces DOP by an example. Section 3 is an overview on the proposed approach for designing a type system for DOP. Section 4 introduces IFJ, 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 for IFJ and IFΔJ, respectively. Section 8 discusses how to enhance early error recognition in delta modules. Related work is discussed in Sect. 9. We conclude by summarizing the paper and outlining some directions for future work in Sect. 10. The appendices contain the proofs of the main results.

A preliminary version of the material presented in this paper appeared in [43]. This paper contains more detailed explanations and examples, an improved version of the IFΔJ calculus (including a more faithful formalization of the `original` construct, which a modified method can use to access the old implementation), a more detailed presentation of the IFJ and IFΔJ calculi, and the proofs of the main results.

2 Delta-oriented programming

In order to illustrate the main concepts of DOP, we use a variant of the *expression product line* (EPL) as described in [36]. The EPL is based on the *expression problem* [50], an extensibility problem that has been proposed as a benchmark for data abstractions' capability 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 the data—Lit, Add, Neg—and the ones concerned with the operations—Eval and Print. Lit and Print are mandatory features. The features Add, Neg and Eval are optional. Figure 1 shows the feature model [28] of the EPL.

The example aims at illustrating the constructs of the IFΔJ calculus presented in Sect. 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 Sect. 4, the encoding in IFJ is straightforward. We refer to Schaefer and Damiani [44] for examples of programming the EPL in DOP that exploit the full JAVA syntax.

2.1 Delta modules

The main concept of DOP are delta modules which are containers of modification operations to an object-oriented program. The modifications may add, remove or modify classes. Modifying a class means changing the super class, adding or removing fields or methods or modifying 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` construct in AHEAD [6], and to the `proceed` construct of AOP, see also Sect. 9). The call `original(···)` expresses a call to the method with the same name before the modifications and is bound at the time the product is generated. Other statements can be introduced before and after a call `original(···)`, wrapping the existing method implementation. This makes `original` different both from `super` in JAVA-like languages (used in a subclass overridden method declaration to access the superclass implementation) and from `inner` in the GBETA language [21] (used in a superclass to invoke possible customizations in subclasses); in fact, `original`, in the resulting generated product, is not a dispatch to a method in another class: it is actually the invocation of the old method which is copied in the generated class.

Delta-oriented programming supports extractive product line development [34] which starts from existing products (called *legacy products*) and turns them into a product line. Listing 1 contains a delta module for introducing an existing 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 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 DOptionalPrint {
  modifies Add {
    modifies String 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

2.2 Delta-oriented product lines

A delta-oriented product line consists of a *code base* and a *product line declaration*. The code base contains a set of delta modules, while the product line declaration creates the connection to the product line variability specified in terms of product features. The product line captures the configuration knowledge [14] of the product line. Listing 5 shows a product line declaration for the EPL. The product line declaration:

– Lists the product features.
– Describes the set of *valid* feature configurations described by the feature model. In the examples, the valid feature configurations are represented by a propositional formula over the set of features. We refer to Batory [5] for a discussion on other possible representations.
– Attaches to each delta module an application condition specifying for which feature configurations the delta module has to be applied. In the examples, the application condition is represented by a propositional constraint over the set of features, given in a **when** clause. 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 feature configurations.
– Fixes the possible application orders of the delta modules by defining a total order on the sets of a partition of the delta modules. Deltas in the same set of the partition can be applied in any order, but the order of the sets must be obeyed. The ordering captures semantic requires-relations that are necessary for the applicability of the delta modules. In the examples, the ordering is represented by writing an ordered list of the delta module sets which are enclosed by `{ .. }` after the keyword **deltas**.

```
features Lit, Add, Neg, Print, Eval
configurations Lit & Print
deltas
  { DLitAddPrint,
    DNeg when Neg }

  { DremAdd when !Add }

  { DLitEval when Eval,
    DAddEval when (Add & Eval),
    DNegEval when (Neg & Eval),
    DNegPrint when Neg,
    DOptionalPrint when (Add & Neg) }
```

Listing 5: Declaration of the EPL

2.3 Product generation

A product is *valid* if it corresponds to a valid feature configuration. In order to obtain a product for a particular feature configuration, the operations specified in the delta modules with satisfied application conditions are applied incrementally to the empty program. Namely, the generation of a product for a given feature configuration consists of two steps, performed automatically:

1. Find all delta modules with a satisfied application condition.
2. Apply the selected delta modules in any linear ordering that respects the total order on the partition of the delta modules. The first delta module is applied to the empty program, the second delta module is applied to the outcome of the application of the first delta module, and so on.

The operations of a delta module are applicable to a program 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, the selected delta modules must be applicable in the given order, otherwise the generation of the product fails. In particular, the first delta module that is applied can only contain additions. The fresh name for the new method introduced by the application of a delta module of name

δ that wraps an existing method of name `m` by using the `original` construct is denoted by `m$`δ.

Listing 6 depicts the product generated when the Lit, Add, Neg, Print and Eval features are selected. Note that in the generated class `Add` the new method `toString$DOptional-Print` has been introduced to implement the call of `original`: it has the same body as the original version of the method `toString` and it is called by the modified version of the method `toString`.

```
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.toString(); }
  int eval() { return value; }
}
class Add extends Exp {
  Exp expr1;
  Exp expr2:
  Add setAdd(Exp a, Exp b) { expr1 = a;  expr2= b; return this; }
  String toString$DOptionalPrint() { return  expr1.toString() + " + " + expr2.toString(); }
  String toString() { return  "(" + toString$DOptionalPrint() + ")"; }
  int eval() { return expr1.eval() + expr2.eval(); }
}
class Neg extends Exp {
  Exp expr;
  Neg setNeg(Exp a) { expr = a; return this; }
  String toString() { return "-" +  expr.toString(); }
  int eval() { return (-1) * expr.eval(); }
}
```

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

The flexibility supported by specifying (via the ordered partition of the delta modules) a set of possible application orders (instead of a single application order) can be exploited to optimize both product type-checking and product generation. This issue, not further discussed in this paper, is investigated in [17].

2.4 Strongly-unambiguous delta-oriented product lines

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 the product line is *unambiguous*, i.e, for every valid feature configuration exactly one product is generated. A product line is *strongly unambiguous* if each set in the partition of the delta modules specified in the product-line declaration is *consistent*, that is, if one delta module in a set adds or removes a class, no other delta module in the same set may add, remove or modify the same class, and the modifications of the same class in different delta modules in the same set have to be disjoint. A strongly unambiguous product line is also unambiguous.

The product line in Listing 5 is strongly unambiguous. Note that the property of being strongly unambiguous is modular, since in order to check it only the consistency of each

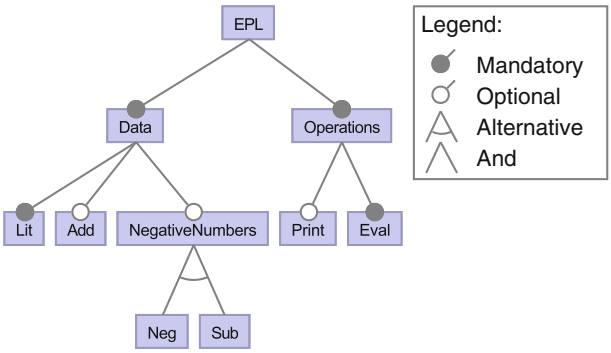

Fig. 2 Feature model for the evolved expression product line

set in the partition of the delta modules specified by the product-line declaration has to be checked.

2.5 Evolution of delta-oriented product lines

Delta-oriented programming also supports reactive product line development [34], starting with an existing product line that is evolved in order to deal with new customer requirements. Consider the example depicted in Listing 5 as the initial product line. Assume now that a new feature Sub needs to be introduced for representing subtraction expressions. In the new product line, the Sub feature becomes an alternative to the Neg feature (i.e., they cannot be both present in the same product). Additionally, the Print feature should become optional and the Eval feature mandatory. The feature model [28] for the evolved product line is given in Fig. 2.

```
features Lit, Add, Neg, Sub, Print, Eval
configurations Lit & Eval & choose1(Neg,Sub)
deltas
  { DLitAddPrint,
    DNeg when Neg,
    DSub when Sub /* new delta module */ }

  { DremAdd when !Add }

  { DremPrintLit when !Print, /* new delta module */
    DremPrintAdd when (!Print & Add) /* new delta module */ }

  { DLitEval when Eval,
    DAddEval when Add,
    DNegEval when Neg,
    DSubEval when Sub, /* new delta module */
    DNegPrint when (Neg & Print),
    DSubPrint when (Sub & Print), /* new delta module */
    DOptionalPrint when (Add & (Neg | Sub) & Print) }
```

Listing 7: Declaration of the evolved EPL

The declaration for the evolved EPL is shown in Listing 7, where the operator **choose1**$(P_1, \ldots, P_n)$ means at most one of the propositions $P_1, \ldots, P_n$ is true (see [5]). The same is captured in the feature diagram (cf. Fig. 2) by introducing the additional (abstract) feature NegativeNumbers to encapsulate the alternative between the features Neg and Sub.

The eight delta modules of the EPL (cf. Listing 5) are reused for the evolved EPL. Moreover, in order to realize the new Sub feature, we have to define delta modules that introduce the corresponding data structure for subtraction and the associated print and the evaluation functionalities. The respective delta modules are shown in Listing 8. We also have to define delta modules for removing the Print feature, shown in Listing 9, since printing is now optional. Note that the evolved EPL is strongly unambiguous.

```
delta DSub {
  adds class Sub extends Exp {
    Exp expr1;
    Exp expr2:
    Sub setSub(Exp a, Exp b) { expr1 = a; expr2= b; return this; }
  }
}

delta DSubPrint {
  modifies Sub {
    adds String toString() { return "(" + expr1.toString() + " - " + expr2.toString() + ")"; }
  }
}

delta DSubEval{
   modifies Sub {
     adds int eval() { return expr1.eval() − expr2.eval(); }
   }
}
```

Listing 8: Delta modules for Sub feature

```
delta DremPrintLit {
  modifies Exp { removes toString }
  modifies Lit { removes toString }
}

delta DremPrintAdd {
  modifies Add { removes toString }
}
```

Listing 9: Delta modules removing the Print feature

```
class Exp extends Object {
  int eval() { return 0; }
}
class Lit extends Exp {
  int value;
  Lit setLit(int n) { value = n; return this; }
  int eval() { return value; }
}
class Sub extends Exp {
  Exp expr1;
  Exp expr2:
  Sub setSub(Exp a, Exp b) { expr1 = a; expr2= b; return this; }
  int eval() { return expr1.eval() − expr2.eval(); }
}
```

Listing 10: Generated code for Lit, Sub and Eval features

Listing 10 depicts the product generated when the Lit, Sub and Eval features are selected. This example shows that DOP supports product line evolution by reusing existing delta modules, by adding new delta modules to implement new product features or to deal with new feature combinations, and by reconfiguring the application conditions and the delta module order in the product line declaration to capture changes in the feature model. Since delta modules can be reused across different product lines, a type system for DOP should support the reuse across different product lines of the types inferred for the delta modules.

3 Type safety and compositional type-checking of DOP product lines

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 analysis techniques to ensure that a product line is type safe can be classified in three main categories [49]:

1. Product-based analyses consider each product variant separately. Product-based analyses can use any standard analysis technique for single products. These analyses work well when relatively few products are generated (as it happens in many practical cases), but are in general infeasible when many products are generated (the number of products of a product line may be exponential in the number of features). According to the product-based approach, the type safety of an SPL could be checked by generating all products and type checking each of them separately. Without a suitable tool support, it can be difficult to trace the source of typing errors in composed code to the originating delta modules.
2. Family-based analyses check the complete code base of the product line in a single run to obtain a result about all possible variants. A family-based product line analysis which relies on a monolithic model of the product line for checking the type safety of FOP product lines has been proposed in [2].
3. Feature-based analyses consider the building blocks of the different product variants (i.e., the feature modules in FOP and the delta modules in DOP) in isolation to derive results on all variants. The results of the analysis of each building block can be reused across different product lines, like the associated building blocks can (cf. Sect. 2.5). Feature-based analyses usually only work for feature-compositional properties, such as syntax checking, or require a final product-based or family-based analysis phase in addition to the feature-based analysis phase. For instance, in the type safety of FOP product lines presented in [19] the feature-based phase generates constraints for every feature module in isolation and the family-based phase checks these constraints for the whole product family.

In this paper, we present a compositional type system for delta-oriented product lines that supports a feature-based phase and a final product-based phase by relying on an abstraction of product generation. The concepts of this approach are demonstrated for IFΔJ (IMPERATIVE FEATHERWEIGHT DELTA JAVA), a core calculus for delta-oriented product lines of JAVA programs based on IFJ (IMPERATIVE FEATHERWEIGHT JAVA), an imperative version of FJ (FEATHERWEIGHT JAVA) [26]. An IFJ program consists of a *class table* $\mathtt{CT}$, i.e., a mapping from class names to class definitions. The approach consists of two technical means:

1. A constraint-based type system for IFJ that infers a set of *class constraints* $\mathscr{C}$ for an IFJ program. A set of class constraints can be understood as a requires interface for a program.

The *class signature table* of CT is a representation of the program without method bodies. It can be understood as a provides interface for a program. The pair $\langle signature(\texttt{CT}), \mathscr{C} \rangle$ is the *type abstraction of the program* CT. The type abstraction of a program can be understood as its provides/requires interface. A checking procedure for the program type abstraction can check the inferred constraints against the class signature table of CT in order to establish whether CT is a well-typed IFJ program.

2. A constraint-based type system for IFΔJ that infers a set of *class-constraint operations* $\mathscr{D}_\delta$ for each delta module δ by considering each delta module separately. A set of class-constraint operations can be understood as a requires interface for a delta module. The signature of a delta module is a representation of the delta module without method bodies (the analogue of the class signature table). It can be understood as a provides interface for a delta module. The *type abstraction of a delta module* δ is the pair $\langle signature(\delta), \mathscr{D}_\delta \rangle$. The type abstraction of a delta module can be understood as its provides/requires interface. The type abstraction of each product can be generated by composing in a delta oriented manner the type abstractions of the delta modules that would be used to generate the product. Namely:
   - The signature of the product (that is, its class signature table) can be generated by composing the signatures of the delta modules. 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 by composing the sets of class-constraint operations of the delta modules.

According to the above explanations, the type safety of each product can be established by relying only on the type abstractions of the delta modules and the product line declaration, by generating and checking the type abstraction of the product (without generating the product).

- The generation of the type abstractions of all the delta modules is a feature-based analysis phase.
- The generation and the checking of the type abstractions of all the products is a product-based analysis phase.

We expect that generating and checking the abstractions of all the products as illustrated above will take less time than generating the implementations of the products and checking them by a JAVA compiler.

4 IFJ

In this section we introduce the syntax and the type system of IFJ (IMPERATIVE FEATHERWEIGHT JAVA), a minimal imperative calculus for JAVA that we use as the underlying calculus to implement single products. The operational semantics and the type soundness of IFJ are given in Appendix A.

IFJ is a variant of FJ [26] that supports modification of fields by field assignment expressions and does not require all the fields to be initialized in a single constructor call. This makes IFJ more suitable than FJ for the formalization of SPLs of JAVA programs, since (as already pointed out in [19]) the fact that FJ requires all the fields to be initialized in a single constructor call, whose parameters have to match the field declarations, makes it difficult to deal with product transformations that add (or remove) fields. The notations and definitions introduced in this section will be intensively used through the rest of the paper.

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

Fig. 3 IFJ: syntax of classes (C ∈ class names, f ∈ field names, m ∈ method names, x ∈ variable names)

4.1 IFJ syntax

The abstract syntax of the IFJ constructs is given in Fig. 3. Following [26], we use the overline notation for possibly empty sequences. For instance, we write "$\bar{\texttt{e}}$" as short for a possibly empty sequence of expressions "$\texttt{e}_1, \ldots, \texttt{e}_n$" and "$\overline{\texttt{MD}}$" as short for a possibly empty sequence of method definitions "$\texttt{MD}_1 \ldots \texttt{MD}_n$" (without commas). The empty sequence is denoted by •. We abbreviate operations on sequences of pairs in similar way, e.g., we write "$\bar{\texttt{C}}\,\bar{\texttt{f}}$" as short for "$\texttt{C}_1\,\texttt{f}_1, \ldots, \texttt{C}_n\,\texttt{f}_n$" and "$\bar{\texttt{C}}\,\bar{\texttt{f}};$" as short for "$\texttt{C}_1\,\texttt{f}_1; \ldots \texttt{C}_n\,\texttt{f}_n;$". 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` {$\overline{\texttt{FD}}$; $\overline{\texttt{MD}}$} consists of its name `C`, its superclass `D` (which must always be specified, even if it is `Object`), a list of field definitions $\overline{\texttt{FD}}$ and a list of method definitions $\overline{\texttt{MD}}$. The fields declared in `C` are added to the ones declared in `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`. In the rest of this section, we assume that a class table `CT` satisfies the following *sanity conditions:*

(i) $\texttt{CT}(\texttt{C}) = \texttt{class C} \ldots$ for every $\texttt{C} \in \mathit{dom}(\texttt{CT})$;
(ii) for every class name `C` (except `Object`) appearing anywhere in `CT`, we have $\texttt{C} \in \mathit{dom}(\texttt{CT})$;
(iii) there are no cycles in the transitive closure of the immediate `extends` relation.

An IFJ program is a class table `CT`.[1] A class definition `CD` can be understood as a mapping from the keyword `extends` to a superclass 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 $\mathit{aDef}(\texttt{C})(\texttt{a})$. For every class `C` in $\mathit{dom}(\texttt{CT})$, the function $\mathit{aDef}(\texttt{C})$ is defined as follows:

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

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

[1] In FJ [26], a program is a pair (CT, 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 e is of type C.

The following example illustrates the IFJ encoding of the EPL product for Lit, Add, Neg, Print and Eval features given in Listing 6 of Sect. 2.3.

Example 1 The sequential composition of two expressions, "$\mathtt{e}_1\,;\,\mathtt{e}_2$", which is not part of the IFJ syntax, can be encoded as follows. Introduce an auxiliary class `Encode` defining a method `sc2C` `C sc2C(Object x1, C x2) { return x2; }` where "`sc`" is short for "sequential composition", "2" is the number of expressions to be composed, and "C" is the type of $\mathtt{e}_2$. Then "$\mathtt{e}_1\,;\,\mathtt{e}_2$" is encoded as `new Encode().sc2C(`$\mathtt{e}_1$`,`$\mathtt{e}_2$`)`

In the following, we assume that the class `Encode` defines suitable methods `sc2Lit` (for the sequential composition of two expressions, where the second one has type `Lit`), `sc3Add` (for the sequential composition of three expressions, where the third one has type `Add`), etc. We also assume a class `Int` for integers, with suitable methods `sum`, `subtract`, etc. The class `String` has the method `concat` for concatenation. The distinction between different `String` literals is immaterial for the purpose of type checking, therefore the occurrences of string values (like `''('')`, `''+''`, etc.) can be encoded without loss of generality by the expression `new String()`. Similarly, the occurrences of integer literals can be encoded by `new Int()`.

According to the above conventions, the classes `Exp`, `Lit`, `Add` and `Neg` in Listing 6 of Sect. 2.3 can be encoded in IFJ as in Listing 11.

```
class Exp extends Object {
  String toString() { return new String(); }
  Int eval() { return new Int(); }
}
class Lit extends Exp {
  Int value;
  Lit setLit(Int n) { return new Encode().sc2Lit(this.value = n, this); }
  String toString() { return this.value.toString(); }
  Int eval() { return this.value; }
}
class Add extends Exp {
  Exp expr1;
  Exp expr2:
  Add setAdd(Exp a, Exp b) { return new Encode().sc3Add(this.expr1=a, this.expr2=b, this); }
  String toString$DOptionalPrint()
    { return this.expr1.toString().concat(new String()).concat(this.expr2.toString()); }
  String toString()
    { return  (new String()).concat(this.toString$DOptionalPrint()).concat(new String()); }
  Int eval() { return this.expr1.eval().sum(this.expr2.eval()); }
}
class Neg extends Exp {
  Exp expr;
  Neg setNeg(Exp a) { return new Encode().sc2Neg(this.expr = a, this); }
  String toString() { return (new String()).concat(this.expr.toString()); }
  Int eval() { return (new Int()).multiply(this.expr.eval()); }
}
```

Listing 11: IFJ encoding of the product for Lit, Add, Neg, Print and Eval features

4.2 IFJ typing

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

$$\begin{array}{ll} \mathtt{CS} ::= \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{C}\ \{\overline{\mathtt{FD}};\ \overline{\mathtt{FD}}\} & \text{class signatures} \\ \mathtt{MH} ::= \mathtt{C}\,\mathtt{m}\,(\overline{\mathtt{C}}\,\overline{\mathtt{x}}) & \text{method headers} \end{array}$$

A *class signature table* CST is a mapping from class names to class signatures. We write *signature*(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 *aType*(C)(a). For every class C in *dom*(CST), the function *aType*(C) is defined as follows:

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

It is possible to check that there are no cycles in the transitive closure of the extends relation by inspection of the class signature table. Moreover, by inspecting a class signature table, it is possible to check, for every class C in *dom*(CST), that the names of the fields defined in C are distinct from the names of the fields inherited from its superclasses, and that 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) $\mathtt{CS(C)} = \mathtt{class\ C} \ldots$ for every $\mathtt{C} \in dom(\mathtt{CS})$;
(ii) for every class name C (except Object) appearing anywhere in CS, we have $\mathtt{C} \in dom(\mathtt{CS})$;
(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 $aType(\mathtt{C}_2)(\mathtt{m})$ is defined then $aType(\mathtt{C}_1)(\mathtt{m}) = aType(\mathtt{C}_2)(\mathtt{m}_{trs})$; and
(v) $\mathtt{C}_1 <: \mathtt{C}_2$ and $\mathtt{C}_1 \neq \mathtt{C}_2$ imply that, for all field names f, if $\mathtt{f} \in dom(\mathtt{CST}(\mathtt{C}_2))$ then $\mathtt{f} \notin dom(\mathtt{CST}(\mathtt{C}_1))$.

In order to type the null value (which is not considered in FJ [26]), the IFJ type system uses the special type $\bot$, that is not a class name, cannot occur in IFJ programs and is a subtype of any other type. We use the metavariable T to denote either a class name or $\bot$.

The IFJ typing rules are given in Fig. 4. 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 is 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 [26]. 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 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, if $\vdash$ CT OK holds, then CT satisfies the sanity conditions for class tables (cf. Sect. 4.1).

Expressions like (C)e where the type of e is not a subtype of C (called *stupid casts* in [26]) or null.f and null.m(···) (that we call *stupid selections*) are ill-typed. Note that, at runtime, an expression without stupid casts and stupid selections may reduce to an expression containing either a stupid cast or a stupid selection. Therefore, the type system for runtime expressions (given in "Appendix A" in order to formulate the type soundness by using standard technique of subject reduction and progress theorems for a small step semantics) contains a rule for

Expression typing $\boxed{\Gamma \vdash \mathtt{e} : \mathtt{T}}$

$$\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \text{(T-Var)}$$

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

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

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

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

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

$$\Gamma \vdash \mathtt{null} : \bot \qquad \text{(T-Null)}$$

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

Method definition typing $\boxed{\mathtt{this} : \mathtt{C} \vdash \mathtt{MD\ OK}}$

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

Class definition typing $\boxed{\vdash \mathtt{CD\ OK}}$

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

Program typing $\boxed{\vdash \mathtt{CT\ OK}}$

$$\frac{\mathit{signature}(\mathtt{CT})\ \text{satisfies the sanity conditions} \qquad \vdash \mathtt{CT}(\mathtt{C})\ \mathtt{OK}^{(\mathtt{C} \in \mathit{dom}(\mathtt{CT}))}}{\vdash \mathtt{CT\ OK}} \qquad \text{(T-Program)}$$

Fig. 4 IFJ: typing rules for expressions, methods, classes and program CT (where CST denotes the class signature table *signature*(CT), and T denotes either a class name or $\bot$)

typing stupid casts and a rule for assigning any type T to the value `null` (so that stupid selection can be typed).

Following [26], we say that a well-typed IFJ program is *cast safe* to mean that it can be typed without using the rule for downcast. Every well-typed IFJ program is literally a well-typed JAVA program.

5 IFΔJ

In this section, we introduce the syntax and the semantics of IFΔJ (IMPERATIVE FEATHERWEIGHT DELTA JAVA), a core calculus for DOP of product lines of JAVA programs based on IFJ.

DMD	::=	delta δ {$\overline{\texttt{CO}}$}	delta modules
CO	::=	adds CD | removes C | modifies C [extending C] { $\overline{\texttt{AO}}$ }	class operations
AO	::=	adds FD | adds MD | removes a | modifies MD	attribute operations

Fig. 5 IFΔJ: syntax of delta modules (where $\delta \in$ delta module names)

5.1 IFΔJ syntax

The abstract syntax of the IFΔJ constructs is given in Fig. 5. The constructs for class definitions CD, field definitions FD and method definitions MD are those of IFJ, given in Fig. 3. The metavariable δ ranges over delta module names.

A delta module definition DMD (see Fig. 5) can be understood as a pair formed by the name δ of the delta module and a mapping from class names to *class operations*. A class operation CO 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:

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

A class-modify operation is defined by possibly changing the super class and by listing a sequence of *attribute operations* AO defining modifications of methods and additions/removals of fields and methods. A class-modify operation CO can be understood as a mapping from the keyword extending to an empty or singleton set of class names and from field/method names to attribute operations. The adds-, removes- and modifies-domain of a class-modify operation CO are defined as follows:

$$\begin{aligned} \mathit{addsDom}(\texttt{CO}) &= \{\texttt{a} \mid \texttt{CO(a)} = \texttt{adds} \cdots \texttt{a} \cdots\} \\ \mathit{removesDom}(\texttt{CO}) &= \{\texttt{a} \mid \texttt{CO(a)} = \texttt{removes a}\} \\ \mathit{modifiesDom}(\texttt{CO}) &= \{\texttt{m} \mid \texttt{CO(m)} = \texttt{modifies} \cdots \texttt{m} \cdots\} \end{aligned}$$

A method-modify operation 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 call this.original($\bar{\texttt{e}}$), which may only occur in the body of the method MD provided by a method-modify operation modifies MD, expresses a call to the method with the same name before the modifications and is bound at the time the product is generated.

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 L $= (\overline{\varphi}, \Phi, \texttt{DMT}, \Delta, \Pi)$ consisting of:

1. the features $\overline{\varphi}$ of the SPL,
2. the set of the valid feature configurations $\Phi \subseteq \mathscr{P}(\overline{\varphi})$,[2]
3. a delta module table DMT containing the delta modules,

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

4. a mapping $\Delta : \Phi \rightarrow \mathscr{P}(dom(\texttt{DMT}))$ determining for which feature configurations a delta module must be applied (which is denoted by the **when** clause in the concrete examples),
5. a totally ordered partition Π of $dom(\texttt{DMT})$, determining the order of delta module application.

```
delta DLitAddPrint {
  adds class Exp extends Object {
    String toString() { return new String(); }
  }
  adds class Lit extends Exp {
    Int value;
    Lit setLit(Int n) { return new Encode().sc2Lit(this.value = n, this); }
    String toString() { return this.value.toString(); }
  }
  adds class Add extends Exp {
    Exp expr1;
    Exp expr2:
    String toString() { return this.expr1.toString().concat(new String()).concat(this.expr2.toString()); }

  }
}

delta DNeg  {
  adds class Neg extends Exp {
    Exp expr;
    Neg setNeg(Exp a) { return new Encode().sc2Neg(this.expr = a, this); }
  }
}

delta DLitEval {
  modifies Exp {
    adds Int eval() { return new Int(); }
  }
  modifies Lit {
    adds Int eval() { return this.value; }
  }
}

delta DAddEval {
  modifies Add {
    adds Int eval() { return this.expr1.eval().sum(this.expr2.eval()); }
  }
}

delta DNegEval{
  modifies Neg {
    adds Int eval() { return (new Int()).multiply(this.expr.eval()); }
  }
}

delta DNegPrint {
  modifies Neg {
    adds String toString() { return (new String()).concat(this.expr.toString());
  }
}

delta DOptionalPrint {
  modifies Add {
    modifies String toString() { return  (new String()).concat(this.original()).concat(new String()); }
  }
}
```

Listing 12: IFΔJ encoding of the delta modules for the feature configuration Lit, Add, Neg, Print, Eval

The 4-tuple $(\overline{\varphi}, \Phi, \Delta, \Pi)$ 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 $\mathtt{L} = (\overline{\varphi}, \Phi, \mathtt{DMT}, \Delta, \Pi)$.

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

Example 2 Consider the EPL example introduced in Sect. 2. Listing 12 illustrates the IFΔJ encoding (according to the conventions introduced in Example 1 of Sect. 4.1) of the delta modules: DLitAddPrint from Listing 1; DLitEval and DAddEval from Listing 2; and DNeg, DNegEval, DNegPrint and DOptionalPrint from Listing 3. These are the delta modules that must be applied when the feature configuration Lit,Add,Neg,Print,Eval is selected.

5.2 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 class-modify operation, if each method or field to be removed exists, if each method to be modified exists and has the same header specified in the method-modify operation, 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 APPLY $(\delta, \mathtt{CT})$, is the class table CT′ defined as follows:

$$\mathtt{CT}'(\mathtt{C}) = \begin{cases} \mathtt{CD} & \text{if } \delta(\mathtt{C}) = \mathtt{adds\ CD} \\ \text{undefined} & \text{if } \delta(\mathtt{C}) = \mathtt{removes\ C} \\ \text{APPLY}_\delta(\delta(\mathtt{C}), \mathtt{CT}(\mathtt{C})) & \text{if } \mathtt{C} \in \mathit{modifiesDom}(\delta) \\ \mathtt{CT}(\mathtt{C}) & \text{otherwise} \end{cases}$$

where $\text{APPLY}_\delta(\delta(\mathtt{C}), \mathtt{CT}(\mathtt{C}))$, the application of the class-modify operation $\delta(\mathtt{C}) = \mathtt{CO}$ to the class definition $\mathtt{CT}(\mathtt{C}) = \mathtt{CD}$, is the class definition CD′ defined as follows (recall that the metavariable AD denotes either a field definition FD or a method definition MD):

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

$$\mathtt{CD}'(\mathtt{a}) = \begin{cases} \mathtt{AD} & \text{if } \mathtt{CO(a)} = \mathtt{adds\ AD} \\ \text{undefined} & \text{if } \mathtt{CO(a)} = \mathtt{removes\ a} \\ \mathtt{MD[a\$}\delta\mathtt{/original]} & \text{if } \mathtt{CO(a)} = \mathtt{modifies\ MD} \\ \mathtt{A\ a}(\bar{\mathtt{A}}\ \bar{\mathtt{x}})\mathtt{\{return\ e;\}} & \text{if } \mathtt{a} = \mathtt{m\$}\delta \text{ for some } \mathtt{m} \text{ such that } \mathtt{CO(m)} = \mathtt{modifies\ MD}, \\ & \quad \mathtt{original} \in \mathtt{MD} \text{ and } \mathtt{CD(m)} = \mathtt{A\ m}(\bar{\mathtt{A}}\ \bar{\mathtt{x}})\mathtt{\{return\ e;\}} \\ \text{undefined} & \text{if } \mathtt{a} = \mathtt{m\$}\cdots \text{ for some } \mathtt{m} \text{ such that} \\ & \quad \mathtt{CO(m)} = \mathtt{removes\ m} \text{ or } (\mathtt{CO(m)} = \mathtt{modifies\ MD} \\ \mathtt{CD(a)} & \text{and } \mathtt{original} \notin \mathtt{MD}) \text{otherwise} \end{cases}$$

The semantics of the original construct is modeled by the third-to-last, the second-to-last and the first-to-last cases of the definition of CD′:

- The third-to-last case specifies that the body of the method m is replaced with the body obtained from the body in the method-modify operation by replacing all the occurrences of the keyword original with the name, denoted but $\mathtt{m\$}\delta$ (where δ is the name of the delta module containing the method-modify operation), of a new method with the original body of the method m.
- The second-to-last case specifies that, if the method body in the method-modify operation contains at least one occurrence of the keyword original, then a new method with the same body of the original method and with name $\mathtt{m\$}\delta$ is introduced.

– The first-to-last case ensures that, if a method `m` is removed or modified without using the `original` construct, then also the auxiliary methods `m$`$\cdots$ that might have been introduced by previously applied delta modules are removed.

An *application order* for the IFΔJ SPL `L` is a total order of its delta modules that is compatible with the ordered partition Π. An application order defines a *product generation mapping*. That is, a partial mapping from each feature configuration $\overline{\psi}$ in Φ to the class table of the product that is obtained by applying the delta modules $\Delta(\overline{\psi})$ 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. The product line is *unambiguous* if all application orders define the same product generation mapping. In an unambiguous SPL, for every feature configuration at most one product implementation is generated.

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.

Example 3 The application to the empty class table of the delta modules in Listing 12 (in the order in which they appear) generates the class table $\mathtt{CT}_{\mathsf{Lit,Add,Neg,Print,Eval}}$ containing the classes given in Listing 11 of Sect. 4.1.

5.3 Well-formed IFΔJ product lines

In this section we formalize in the context of IFΔJ the notion of strongly-unambiguous SPL (informally introduced in Sect. 2.4) and the notion of type-safe SPL (informally introduced in Sect. 3).

An IFΔJ SPL is *strongly unambiguous* if every set S of delta modules in Π is *consistent*. That is, if no class added or removed in a delta module of S is added, removed or modified in another delta module of S, and for every class modified in more than one delta module of S, its direct superclass is changed at most by one delta clause and the fields and methods added, modified or removed are distinct.

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. 5, by replacing class definitions (`CD`) with class signatures (`CS`) and by replacing method definitions (`MD`) with method headers (`MH`). We write *signature*(δ) to denote the signature of the delta module δ.

If a product line is strongly unambiguous, then it is also unambiguous. In a strongly unambiguous product line, two delta modules that modify the same method cannot be placed in the same set of the partition even if they are never applied together. The property of being a strongly-unambiguous product line is modular. It can be efficiently checked by only relying on delta module signatures and the partition Π and it is preserved by: signature preserving alterations to the delta modules, shrinking of the partition Π, changes to the application conditions and changes to the set of valid feature configurations.

An IFΔJ SPL is *well formed* if the following conditions hold:

1. it is strongly unambiguous, and
2. it is type safe, that is, all valid products are well-typed IFJ programs (this implies that the product generation mapping is total).

Class constraints
(a class constraint is denoted by *cc*, a set of class constraints is denoted by $\mathscr{C}$):

`C` **with** $\mathscr{M}$ class `C` has the set of method constraints $\mathscr{M}$

Method constraints
(a method constraint is denoted by *mc*, a set of method constraints is denoted by $\mathscr{M}$):

`m` **with** $\mathscr{E}$ method `m` has the set of expression constraints $\mathscr{E}$

Fig. 6 IFJ: syntax of class constraints

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 Sect. 4. The constraint-based type system for IFJ infers a *class constraint* for each class definition in the program being typed. The set of class constraints inferred for a program `CT` can then be checked against the class signature table *signature*(`CT`) in order to establish whether `CT` is a well-typed IFJ program.

The constraint-based type system for IFJ extracts from a program `CT` the information relevant to typing that is not present in its class signature table *signature*(`CT`) and encodes them by constraints. The encoding is quite straightforward. Note that the constraints and the constraint-based typing rules have been designed to be directly embedded in the constraint-based type system IFΔJ (in Sect. 7). Namely:

- The inferred constraints are organized in a two-level hierarchy, corresponding to the structure of the class table of the IFJ program:
  1. 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
  2. each method constraint consists of the name of the respective method and of the set of *expression constraints* inferred for the body of the method.

  The hierarchical organization of the constraints for a program supports the generation of the constraint of a product starting from the constraints operation inferred (by the constraint-based type system for IFΔJ) for the delta modules that would be used to generate the product (cf. Sect. 3).
- The typing rule for classes generate each class constraint by analyzing each class in isolation from the other classes in the program, and the rule for methods generate each method constraint by analyzing each method in isolation from the other methods in the class.

6.1 Constraints and expression constraints checking

The syntax of class constraints and method constraints is given in Fig. 6. A set of class constraints $\mathscr{C}$ can be understood as a mapping from class names to class constraints, and a set of method constraint $\mathscr{M}$ can be understood as a mapping from method names to method constraints. The hierarchical organization of the constraints is immaterial for checking their satisfaction (i.e, only expression constraints have to be checked).

The syntax of expression constraints is given in Fig. 7. Expression constraints 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

Expression constraints (a set of expression constraints is denoted by $\mathscr{E}$):

$\mathbf{class}(\mathtt{C})$	class $\mathtt{C}$ 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 $\mathtt{C}$
$\mathbf{field}(\eta,\mathtt{f},\alpha)$	class η must define or inherit field $\mathtt{f}$ of type α
$\mathbf{meth}(\eta,\mathtt{m},\overline{\alpha}\to\beta)$	class η must define or inherit method $\mathtt{m}$ of type $\overline{\alpha}\to\beta$

Fig. 7 IFJ: syntax and (informal) meaning of expression constraints, where: α and β denote type variables; η denotes either a class name or a type variable; and τ denotes either the type $\bot$, or a class name, or a type variable

$$\mathtt{CST} \models \emptyset \Rightarrow [\,] \qquad \frac{\mathtt{C} \in \mathit{dom}(\mathtt{CST}) \quad \mathtt{CST} \models \mathscr{E} \Rightarrow \mathbf{s}}{\mathtt{CST} \models \{\mathbf{class}(\mathtt{C})\} \uplus \mathscr{E} \Rightarrow \mathbf{s}} \qquad \frac{\mathtt{T} <: \mathtt{C} \quad \mathtt{CST} \models \mathscr{E} \Rightarrow \mathbf{s}}{\mathtt{CST} \models \mathbf{subtype}(\mathtt{T},\mathtt{C}) \uplus \mathscr{E} \Rightarrow \mathbf{s}}$$

$$\frac{\mathtt{T} <: \mathtt{C} \quad \mathtt{CST} \models \mathscr{E} \Rightarrow \mathbf{s}}{\mathtt{CST} \models \{\mathbf{cast}(\mathtt{C},\mathtt{T})\} \uplus \mathscr{E} \Rightarrow \mathbf{s}} \qquad \frac{\mathtt{C} <: \mathtt{T} \quad \mathtt{C} \neq \mathtt{T} \quad \mathtt{CST} \models \mathscr{E} \Rightarrow \mathbf{s}}{\mathtt{CST} \models \{\mathbf{cast}(\mathtt{C},\mathtt{T})\} \uplus \mathscr{E} \Rightarrow \mathbf{s}}$$

$$\frac{\mathit{aType}(\mathtt{C})(\mathtt{f}) = \mathtt{A} \quad \mathtt{CST} \models \mathscr{E}[\mathtt{A}/\alpha] \Rightarrow \mathbf{s}}{\mathtt{CST} \models \{\mathbf{field}(\mathtt{C},\mathtt{f},\alpha)\} \uplus \mathscr{E} \Rightarrow \mathbf{s} \circ [\mathtt{A}/\alpha]}$$

$$\frac{\mathit{aType}(\mathtt{C})(\mathtt{m}) = \mathtt{A}_1 \cdots \mathtt{A}_n \to \mathtt{A}_0 \quad \mathtt{CST} \models \mathscr{E}[\mathtt{A}_0 \cdots \mathtt{A}_n/\alpha_0 \cdots \alpha_n] \Rightarrow \mathbf{s}}{\mathtt{CST} \models \{\mathbf{meth}(\mathtt{C},\mathtt{m},\alpha_1 \cdots \alpha_n \to \alpha_0)\} \uplus \mathscr{E} \Rightarrow \mathbf{s} \circ [\mathtt{A}_0 \cdots \mathtt{A}_n/\alpha_0 \cdots \alpha_n]}$$

Fig. 8 IFJ: Checking rules for satisfaction of expression constraints w.r.t. a class signature table

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.

The checking judgment for expression constraints is $\mathtt{CST} \models \mathscr{E} \Rightarrow \mathbf{s}$, to be read "the constraints in the set of expression constraints $\mathscr{E}$ are satisfied with respect to the class signature table $\mathtt{CST}$ modulo the substitution $\mathbf{s}$". We write $\mathtt{CST} \models \mathscr{E}$ to mean that $\mathtt{CST} \models \mathscr{E} \Rightarrow \mathbf{s}$ holds for some substitution $\mathbf{s}$. The associated rules are given in Fig. 8 where $\uplus$ denotes the disjoint union of sets of constraints and $\circ$ denotes the composition of substitutions. The rules are almost self-explanatory, according to the informal meaning of the expression constraints given in Fig. 7. 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 type variables only. It causes the instantiation of all the type variables occurring in the third argument.

We say that a set of expression constraints is *cast safe with respect to a class signature table* $\mathtt{CST}$ 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 judgment for programs is $\vdash \mathtt{CT} : \mathscr{C}$, to be read "program $\mathtt{CT}$ has the class constraints $\mathscr{C}$". The constraint-based typing rules for IFJ expressions, methods, classes and programs are given in Fig. 9. The rules (CT- FIELD) and (CT- INVK) are

Expression typing $\boxed{\Gamma \vdash \mathtt{e} : \tau \mid \mathscr{E}}$

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

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

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

$$\Gamma \vdash \mathtt{new\ C}() : \mathtt{C} \mid \{\mathbf{class}(\mathtt{C})\} \qquad \text{(CT-NEW)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \tau \mid \mathscr{E}}{\Gamma \vdash (\mathtt{C})\mathtt{e} : \mathtt{C} \mid \mathscr{E} \cup \{\mathbf{cast}(\mathtt{C}, \tau)\}} \qquad \text{(CT-CAST)}$$

$$\Gamma \vdash \mathtt{null} : \bot \mid \emptyset \qquad \text{(CT-NULL)}$$

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

Method definition typing $\boxed{\mathtt{this} : \mathtt{C} \vdash \mathtt{MD} : \mathit{mc}}$

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

Class definition typing $\boxed{\vdash \mathtt{CD} : \mathit{cc}}$

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

Program typing $\boxed{\vdash \mathtt{CT} : \mathscr{C}}$

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

Fig. 9 IFJ: constraint-based typing rules for expressions, methods, classes and programs

the only rules that create type variables. The type variables α created by rule (CT- FIELD) occur in the third argument of the expression 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 expression constraint $\mathbf{meth}(\eta, \mathtt{m}, \alpha_1 \ldots \alpha_n \to \beta)$. Therefore, the checking rules for expression constraints (given in Sect. 6.1) can be applied by considering the constraints in the order in which they are created. Recall that: (i) the check of the constraints $\mathbf{field}(\cdot, \cdot, \cdot)$ and $\mathbf{meth}(\cdot, \cdot, \cdot)$ can be performed only when their first arguments are class names and their third arguments contain type variables only; and (ii) performing the check causes the instantiation of all the type variables occurring in the constraint. It is worth observing that trying to check the constraints in a different order cannot cause a different instantiation of any type variable.

Some expressions are recognized as ill-typed during constraint generation. In particular, occurrences of variables x that are not declared as method parameters (according to rules

(CT- METHOD) and (CT- VAR)) and stupid selections, i.e., expressions like `null.f` and `null.m(⋯)` (according to rules (CT- NULL), (CT- FIELD) and (CT- INVK)). Moreover, during constraint generation it could be possible to detect type errors related to the use of classes not belonging to the product-line (and used by the products). In particular, in a full-fledged language, the type errors related to the use of primitive types (like `int`) or standard library classes (like `String`).

The following example illustrates the constraint-based type system by considering the class `Add` in Listing 11 (see Example 1 of Sect. 4.1).

Example 4 The constraint inferred for the method `setAdd` of class `Add` in Listing 11

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

is

$$\texttt{setAdd}\ \textbf{with}\ \left\{\begin{array}{l}\textbf{class}(\texttt{Encode}),\ \textbf{meth}(\texttt{Encode},\texttt{sc3C},(\texttt{Object},\texttt{Object},\texttt{Add}\to\texttt{Add})),\\ \quad\textbf{field}(\texttt{Add},\texttt{expr1},\alpha_1),\ \textbf{subtype}(\texttt{Exp},\alpha_1),\ \textbf{subtype}(\alpha_1,\texttt{Object}),\\ \quad\textbf{field}(\texttt{Add},\texttt{expr2},\alpha_2),\ \textbf{subtype}(\texttt{Exp},\alpha_2),\ \textbf{subtype}(\alpha_2,\texttt{Object}),\\ \quad\textbf{subtype}(\texttt{Add},\texttt{Add})\end{array}\right\}$$

Some optimizations (not considered in this paper) are possible. Constraints like **subtype** (`...,Object`) and **subtype** (`Add,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:

$$\texttt{setAdd}\ \textbf{with}\ \left\{\begin{array}{l}\textbf{field}(\texttt{Add},\texttt{expr1},\alpha_1),\ \textbf{subtype}(\texttt{Exp},\alpha_1),\\ \textbf{field}(\texttt{Add},\texttt{expr2},\alpha_2),\ \textbf{subtype}(\texttt{Exp},\alpha_2)\end{array}\right\}$$

The constraint inferred for the method `eval` of class `Add` in Listing 11

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

is

$$\texttt{eval}\ \textbf{with}\ \left\{\begin{array}{l}\textbf{field}(\texttt{Add},\texttt{expr1},\alpha'),\ \textbf{meth}(\alpha',\texttt{eval},(\bullet\to\beta')),\\ \textbf{meth}(\beta',\texttt{sum},(\beta'''\to\beta)),\\ \quad\textbf{field}(\texttt{Add},\texttt{expr2},\alpha''),\ \textbf{meth}(\alpha'',\texttt{eval},(\bullet\to\beta'')),\ \textbf{subtype}(\beta'',\beta''')\\ \textbf{subtype}(\beta,\texttt{Int})\end{array}\right\}$$

Assuming that `Int` is a standard library final class, it would be possible to infer a simpler constraint (namely, replace β by `Int` and drop **subtype**(β, `Int`)). Further optimizations are possible in the presence of primitive types (not formalized in IFJ). For instance, given the version of method `eval` of class `Add` in Listing 6 (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

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

The constraint inferred for the method `toString$DOptionalPrint()` of class `Add` in Listing 11

```
String toString$DOptionalPrint() {
   return  this.expr1.toString().concat(new String()).concat(this.expr2.toString()); }
```

is

$$\texttt{toString\$DOptionalPrint}\ \textbf{with}\ \left\{\begin{array}{l}\textbf{field}(\texttt{Add},\texttt{expr1},\gamma_1),\ \textbf{meth}(\gamma_1,\texttt{toString},(\bullet\to\sigma_1)),\\ \textbf{meth}(\sigma_1,\texttt{concat},(\sigma_2\to\sigma_3)),\\ \quad\textbf{class}(\texttt{String}),\ \textbf{subtype}(\texttt{String},\sigma_2),\\ \textbf{meth}(\sigma_3,\texttt{concat},(\sigma_4\to\sigma_5)),\\ \quad\textbf{field}(\texttt{Add},\texttt{expr2},\gamma_2),\ \textbf{meth}(\gamma_2,\texttt{toString},(\bullet\to\sigma_6)),\ \textbf{subtype}(\sigma_6,\sigma_4),\\ \textbf{subtype}(\sigma_5,\texttt{String})\end{array}\right\}$$

Assuming that `String` is a standard library final class and that every class has a method `toString` with type $\bullet\to$ `String`, it would be possible to infer the simpler constraint

$$\texttt{Exp}\ \textbf{with}\ \emptyset$$

$$\texttt{Lit}\ \textbf{with}\left\{\begin{array}{l}\texttt{setLit}\ \textbf{with}\ \{\ \textbf{field}(\texttt{Lit},\texttt{value},\texttt{int})\ \}\\ \texttt{toString}\ \textbf{with}\ \{\ \textbf{field}(\texttt{Lit},\texttt{value},\texttt{int})\ \}\\ \texttt{eval}\ \textbf{with}\ \{\ \textbf{field}(\texttt{Lit},\texttt{value},\texttt{int})\ \}\end{array}\right\}$$

$$\texttt{Add}\ \textbf{with}\left\{\begin{array}{l}\texttt{setAdd}\ \textbf{with}\ \left\{\begin{array}{l}\textbf{field}(\texttt{Add},\texttt{expr1},\alpha_1),\ \textbf{subtype}(\texttt{Exp},\alpha_1),\\ \textbf{field}(\texttt{Add},\texttt{expr2},\alpha_2),\ \textbf{subtype}(\texttt{Exp},\alpha_2)\end{array}\right\}\\ \texttt{toString\$DOptionalPrint}\ \textbf{with}\ \{\ \textbf{field}(\texttt{Add},\texttt{expr1},\gamma_1),\textbf{field}(\texttt{Add},\texttt{expr2},\gamma_2)\ \}\\ \texttt{toString}\ \textbf{with}\ \{\ \textbf{meth}(\texttt{Add},\texttt{toString\$DOptionalPrint},(\bullet\to\texttt{String}))\ \}\\ \texttt{eval}\ \textbf{with}\left\{\begin{array}{l}\textbf{field}(\texttt{Add},\texttt{expr1},\alpha'),\ \textbf{meth}(\alpha',\texttt{eval},(\bullet\to\texttt{int})),\\ \textbf{field}(\texttt{Add},\texttt{expr2},\alpha''),\ \textbf{meth}(\alpha'',\texttt{eval},(\bullet\to\texttt{int}))\end{array}\right\}\end{array}\right\}$$

$$\texttt{Neg}\ \textbf{with}\left\{\begin{array}{l}\texttt{setNeg}\ \textbf{with}\ \{\ \textbf{field}(\texttt{Neg},\texttt{expr},\beta_1),\ \textbf{subtype}(\texttt{Exp},\beta_1),\ \}\\ \texttt{toString}\ \textbf{with}\ \{\ \textbf{field}(\texttt{Neg},\texttt{expr},\beta_2),\ \}\\ \texttt{eval}\ \textbf{with}\ \{\ \textbf{field}(\texttt{Neg},\texttt{expr},\beta'),\ \textbf{meth}(\beta',\texttt{eval},(\bullet\to\texttt{int})),\ \}\end{array}\right\}$$

Fig. 10 Constraints inferred for the classes in Listing 11

$$\texttt{toString\$DOptionalPrint}\ \textbf{with}\ \{\ \textbf{field}(\texttt{Add},\texttt{expr1},\gamma_1),\textbf{field}(\texttt{Add},\texttt{expr2},\gamma_2)\ \}$$

The constraint inferred for the method `toString` of class `Add` in Listing 11

```
String toString() {
   return  (new String()).concat(this.toString$DOptionalPrint()).concat(new String()); }
```

is

$$\texttt{toString}\ \textbf{with}\left\{\begin{array}{l}\textbf{class}(\texttt{String}),\ \textbf{meth}(\texttt{String},\texttt{concat},(\sigma'\to\sigma'')),\\ \quad\textbf{meth}(\texttt{Add},\texttt{toString\$DOptionalPrint},(\bullet\to\sigma)),\ \textbf{subtype}(\sigma,\sigma'),\\ \textbf{meth}(\sigma'',\texttt{concat},(\sigma'''\to\sigma'''')),\\ \quad\textbf{subtype}(\texttt{String},\sigma'''),\\ \textbf{subtype}(\sigma'''',\texttt{String})\end{array}\right\}$$

Assuming that `String` is a standard library final class and that every class has a method `toString` with type $\bullet\to\texttt{String}$, it would be possible to infer the simpler constraint

$$\texttt{toString}\ \textbf{with}\ \{\ \textbf{meth}(\texttt{Add},\texttt{toString\$DOptionalPrint},(\bullet\to\texttt{String}))\ \}$$

Figure 10 shows the constraints inferred for the classes in Listing 11 (for the sake of readability, the simplified version of the constraints is used).

6.3 Properties

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 expression constraints $\text{FLAT}(\mathscr{C})$. It is defined as follows.

$$\begin{aligned}\text{FLAT}\{\texttt{C}_1\ \textbf{with}\ \mathscr{M}_1,\ \ldots,\ \texttt{C}_n\ \textbf{with}\ \mathscr{M}_n\} &= \cup_{i\in\{1,\ldots,n\}}\text{FLAT}(\mathscr{M}_i)\\ \text{FLAT}\{\texttt{m}_1\ \textbf{with}\ \mathscr{E}_1,\ \ldots,\ \texttt{m}_n\ \textbf{with}\ \mathscr{E}_n\} &= \cup_{i\in\{1,\ldots,n\}}\mathscr{E}_i\end{aligned}$$

The following theorem states that the constraint-based type system is sound and complete with respect to the IFJ type system given in Sect. 4. The proof is given in "Appendix B".

Theorem 1 (Soundness and completeness of IFJ constraint-based typing) *Let* CT *be a IFJ program and* $\texttt{CST} = \textit{signature}(\texttt{CT})$.

(Soundness) Let CST *satisfy the sanity conditions for class signature tables,* $\vdash \texttt{CT} : \mathscr{C}$ *and* $\texttt{CST} \models \text{FLAT}(\mathscr{C})$. *Then*

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

Class-constraints operations
(a class-constraint operation is denoted by *cco*, a set of class-constraint operations is denoted by $\mathscr{D}$):

`adds C` **with** $\mathscr{M}$	add the constraint "`C` **with** $\mathscr{M}$"
`removes C`	remove constraint "`C` **with** $\cdots$"
`modifies C` **with** $\mathscr{O}$	change the constraint "`C` **with** $\mathscr{M}$" by applying to it "`modifies C` **with** $\mathscr{O}$"

Method-constraint operations
(a method-constraint operation is denoted by *mco*,
a set of method-constraint operations is denoted by $\mathscr{O}$,
a singleton or empty set of method-constraint operations is denoted by $\mathscr{S}$):

`adds m` **with** $\mathscr{E}$	add the constraint "`m` **with** $\cdots$"
`removes m`	remove constraint "`m` **with** $\cdots$"
`replaces m` **with** $\mathscr{E}'$	change constraint "`m` **with** $\mathscr{E}$" into "`m` **with** $\mathscr{E}'$"
`wraps m` **with** $\mathscr{E}'$	change constraint "`m` **with** $\mathscr{E}$" into "`m` **with** $\mathscr{E}'[\mathtt{m\$}\delta/\mathtt{original}]$" and add constraint "`m$`δ **with** $\mathscr{E}$" (where δ is the delta module containing the clause for which the constraint has been inferred)

Fig. 11 IFΔJ: syntax of class-constraint operations

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

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

The constraint-based type system for IFJ is not interesting in its own (cf. Sect. 3). However, it soundness and completeness plays an important role in the proof of soundness and completeness of the constraint-based type system for IFΔJ.

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 valid feature 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 $\Delta(\overline{\psi})$ 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 class-constraint operations* $\mathscr{D}$. For each valid feature configuration $\overline{\psi}$, the set of class constraints $\mathscr{C}_{\overline{\psi}}$ of the product $\mathtt{CT}_{\overline{\psi}}$ can be generated by applying the sets of class-constraint operations inferred for the delta modules in $\Delta(\overline{\psi})$ to the empty set of class constraints. Therefore, the type safety of a product line can be established (by relying only on the signatures of the delta modules, the sets of class-constraint operations inferred for the delta modules, and the product line declaration) without reinspecting the delta modules and without generating the products.

7.1 Constraint-based typing rules for delta modules

The typing rules infer for a delta module a set of class-constraints operations $\mathscr{D}$, namely a class-constraint operation for each class operation in the delta module. The syntax of the class-constraint operations is given in Fig. 11. A class-constraint operation can be an *add*, a *remove* or a *modify* operation. A class-constraint-add operation consists of the keyword `adds` followed by a class constraint (defined in Fig. 6). A class-constraint-remove operation is a

Method-operation typing $\boxed{\texttt{this} : \texttt{C} \vdash \texttt{AO} : \mathscr{S}}$

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

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

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

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

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

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

Class-operation typing $\boxed{\vdash \texttt{CO} : \mathit{cco}}$

$$\frac{\vdash \texttt{CD} : \texttt{C}\ \textbf{with}\ \mathscr{M}}{\vdash \texttt{adds CD} : \texttt{adds C}\ \textbf{with}\ \mathscr{M}} \qquad \text{(CT-C-ADDC)}$$

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

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

Delta-module typing $\boxed{\vdash \texttt{DMD} : \mathscr{D}}$

$$\frac{\forall i \in 1..n, \quad \vdash \texttt{CO}_i : \mathit{cco}_i}{\vdash \texttt{delta}\ \delta\ \{\texttt{CO}_1 \ldots \texttt{CO}_n\} : \{\mathit{cco}_1, \ldots, \mathit{cco}_n\}} \qquad \text{(CT-DELTA)}$$

Fig. 12 IFΔ J: constraint-based typing rules for method operations, class operations and delta modules

class-remove operation `removes C`. Each class-constraint-modify operation consists of the name of the subject class `C` and of a set of *method-constraint operations*. A method-constraint operation can be an *add*, a *remove*, a *replace* or a *wrap* operation, described as follows.

- A method-constraint-add operation consists of the keyword `adds` followed by a method constraint (defined in Fig. 6).
- A method-constraint-remove operation is a method-remove operation `removes m`.
- A method-constraint-wrap/replace operation consists of the keyword `replaces/wraps` followed by a method constraint (defined in Fig. 6). Method-constraint-wrap operations are inferred for method-modify operations containing `original`, while method-constraint-replace operations are inferred for method-modify operations not containing `original`.

Thus, a set of class-constraint operations $\mathscr{D}$ can be understood as a mapping from class names to class-constraint operations, and a class-constraint-modify operation can be understood as a mapping from method names to method-constraint operations.

The constraint-based typing judgment for a delta module is $\vdash \texttt{delta}\ \delta\ \cdots : \mathscr{D}$, to be read as "the delta module δ has class-constraints operations $\mathscr{D}$". The constraint-based typing rules for IFΔJ attribute operations, class operations and delta modules are given in Fig. 12. Most of the rules are self-explanatory, according to the meaning of method-constraint operations and class-constraint operations illustrated above. The rules for the attribute operations for

$$\mathscr{D}_{\texttt{DLitAddPrint}} = \left\{ \begin{array}{l} \texttt{adds Exp with } \emptyset, \\ \texttt{adds Lit with } \left\{ \begin{array}{l} \texttt{setLit with } \{\, \textbf{field}(\texttt{Lit}, \texttt{value}, \texttt{int}) \,\}, \\ \texttt{toString with } \{\, \textbf{field}(\texttt{Lit}, \texttt{value}, \texttt{int}) \,\}, \\ \texttt{eval with } \{\, \textbf{field}(\texttt{Lit}, \texttt{value}, \texttt{int}) \,\} \end{array} \right\}, \\ \texttt{adds Add with } \{\, \texttt{toString with } \{\, \textbf{field}(\texttt{Add}, \texttt{expr1}, \gamma_1), \textbf{field}(\texttt{Add}, \texttt{expr2}, \gamma_2) \,\} \,\} \end{array} \right\}$$

$$\mathscr{D}_{\texttt{DNeg}} = \{\, \texttt{adds Neg with } \{\, \texttt{setNeg with } \{\, \textbf{field}(\texttt{Neg}, \texttt{expr}, \beta_1), \textbf{subtype}(\texttt{Exp}, \beta_1) \,\} \,\} \,\}$$

$$\mathscr{D}_{\texttt{DLitEval}} = \left\{ \begin{array}{l} \texttt{modifies Exp with } \{\, \texttt{adds eval with } \emptyset \,\}, \\ \texttt{modifies Lit with } \{\, \texttt{adds eval with } \{\, \textbf{field}(\texttt{Lit}, \texttt{value}, \texttt{int}) \,\} \,\} \end{array} \right\}$$

$$\mathscr{D}_{\texttt{DAddEval}} = \left\{ \texttt{modifies Add with } \left\{ \texttt{adds eval with } \left\{ \begin{array}{l} \textbf{field}(\texttt{Add}, \texttt{expr1}, \alpha'), \textbf{meth}(\alpha', \texttt{eval}, (\bullet \to \texttt{int})), \\ \textbf{field}(\texttt{Add}, \texttt{expr2}, \alpha''), \textbf{meth}(\alpha'', \texttt{eval}, (\bullet \to \texttt{int})) \end{array} \right\} \right\} \right\}$$

$$\mathscr{D}_{\texttt{DNegEval}} = \{\, \texttt{modifies Neg with } \{\, \texttt{adds eval with } \{\, \textbf{field}(\texttt{Neg}, \texttt{expr}, \beta'), \textbf{meth}(\beta', \texttt{eval}, (\bullet \to \texttt{int})) \,\} \,\} \,\}$$

$$\mathscr{D}_{\texttt{DNegPrint}} = \{\, \texttt{modifies Neg with } \{\, \texttt{adds toString with } \{\, \textbf{field}(\texttt{Neg}, \texttt{expr}, \beta_2) \,\} \,\} \,\}$$

$$\mathscr{D}_{\texttt{DOptionalPrint}} = \{\, \texttt{modifies Add with } \{\, \texttt{wraps toString with } \{\, \textbf{meth}(\texttt{Add}, \texttt{original}, (\bullet \to \texttt{String})) \,\} \,\} \,\}$$

Fig. 13 Sets of class-constraints operations inferred for the delta modules in Listing 12

adding and removing a field [(CT- S- ADDF) and (CT- S- REMF), respectively] generate the empty set of constraint operations, 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 Fig. 9. The rule (CT- C- MODC) has an optional part (enclosed in square brackets) to cope with the fact that the `extending`part of a class-modify operation is optional.

Example 5 Figure 13 shows the constraints inferred for the delta modules in Listing 12. For sake of readability, we use the version of the constraints simplified according to the explanation given in Example 4.

7.2 Generation of the class signature tables and the class constraints of the products

The application of a delta module signature to a class signature table, denoted by SIGNAPPLY(DMS, CST), performs the alterations specified in DMS to CST. We do not present the formal definition of SIGNAPPLY(DMS, CST) since it is a straightforward abstraction of the application of a delta module to a class table presented in Sect. 5.2. A delta module signature is *applicable* to a class signature table if each class signature to be removed or modified exists and, for every class-modify operation, if each method header or field to be removed exists, if the header of each method to be modified exists and is the same header specified in the method-modify operation, and if each class signature, method header or field to be added does not exist (cf. the definition of delta module applicable to a class table, given at the beginning of Sect. 5.2).

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). The proof is straightforward, by case distinction on the definitions of delta module applicable to a class table, delta module signature applicable to a class table signature, and on the definitions of the functions *signature*, SIGNAPPLY and APPLY.

Proposition 1 1. *The delta module δ is applicable to the class table* CT *if and only if signature(δ) is applicable to signature*(CT).

2. *If the delta module* δ *is applicable to the class table* $\mathtt{CT}$*, then* $\textsc{signApply}(\mathit{signature}(\delta), \mathit{signature}(\mathtt{CT})) = \mathit{signature}(\textsc{apply}(\delta, \mathtt{CT}))$.

For each valid feature configuration $\overline{\psi}$ of a strongly unambiguous product line, we write $\mathtt{CST}_{\overline{\psi}}$ to denote the class signature table obtained by applying the signatures of the delta modules $\Delta(\overline{\psi})$ to the empty class signature table in any linear ordering that respects the total order on the partition of the delta modules specified in the product line declaration. The following corollary states that $\mathtt{CST}_{\overline{\psi}}$ is indeed the class signature table of the product $\mathtt{CT}_{\overline{\psi}}$.

Corollary 1 (of Proposition 1) *Let* $\mathtt{L}$ *be a strongly unambiguous IFΔJ SPL and* $\overline{\psi} \in \Phi$.

1. *The product* $\mathtt{CT}_{\overline{\psi}}$ *is defined if and only if the class signature table* $\mathtt{CST}_{\overline{\psi}}$ *is defined.*
2. $\mathtt{CST}_{\overline{\psi}} = \mathit{signature}(\mathtt{CT}_{\overline{\psi}})$.

Therefore, the class signature table of any product can be generated without generating the product.

Given a delta module δ and a class table $\mathtt{CT}$ such that δ is applicable to $\mathtt{CT}$, the result of the application of the set of class-constraint operations $\mathscr{D}$ of δ to the set of class constraints $\mathscr{C}$ of $\mathtt{CT}$ such that, denoted by $\textsc{consApply}_\delta(\mathscr{D}, \mathscr{C})$, is the set of class constraints $\mathscr{C}'$ defined as follows:

$$\mathscr{C}'(\mathtt{C}) = \begin{cases} cc & \text{if } \mathscr{D}(\mathtt{C}) = \mathtt{adds}\ cc \\ \text{undefined} & \text{if } \mathscr{D}(\mathtt{C}) = \mathtt{removes\ C} \\ \textsc{consApply}_\delta(\mathscr{D}(\mathtt{C}), \mathscr{C}(\mathtt{C})) & \text{if } \mathscr{D}(\mathtt{C}) = \mathtt{modifies\ C} \cdots \\ \mathscr{C}(\mathtt{C}) & \text{otherwise} \end{cases}$$

where the application of the class-constraint-modify operation $cco = \mathtt{modifies\ C}\ \mathbf{with}\ \mathscr{O} = \mathscr{D}(\mathtt{C})$ to the class-constraint $cc = \mathtt{C}\ \mathbf{with}\ \mathscr{M} = \mathscr{C}(\mathtt{C})$, denoted by $\textsc{consApply}_\delta(cco, cc)$, is the class-constraint $cc' = \mathtt{C}\ \mathbf{with}\ \mathscr{M}'$ defined as follows:

$$cc'(\mathtt{m}) = \begin{cases} mc & \text{if } cco(\mathtt{m}) = \mathtt{adds}\ mc \text{ or } cco(\mathtt{m}) = \mathtt{replaces}\ mc \\ \text{undefined} & \text{if } cco(\mathtt{m}) = \mathtt{removes\ m} \\ \mathtt{m}\ \mathbf{with}\ (\mathscr{E}[\mathtt{m\$}\delta/\mathtt{original}]) & \text{if } cco(\mathtt{m}) = \mathtt{wraps\ m}\ \mathbf{with}\ \mathscr{E} \\ \mathtt{m}\ \mathbf{with}\ \mathscr{E} & \text{if } \mathtt{m} = \mathtt{m}'\mathtt{\$}\delta \text{ for some } \mathtt{m}' \text{ such that } cco(\mathtt{m}') = \mathtt{wraps\ m}'\ \mathbf{with}\ \mathscr{E}' \text{ and } cc(\mathtt{m}') = \mathtt{m}'\ \mathbf{with}\ \mathscr{E} \\ \text{undefined} & \text{if } \mathtt{m} = \mathtt{m}'\mathtt{\$}\cdots \text{ for some } \mathtt{m}' \text{ such that } cco(\mathtt{m}') = \mathtt{removes\ m}' \text{ or } cco(\mathtt{m}') = \mathtt{replaces}\ \cdots \\ cc(\mathtt{m}) & \text{otherwise} \end{cases}$$

The application of a set of class-constraint operations to a set of class constraints mimics the application of a delta module to class table. The semantics of the `original` construct is modeled by the the third, the fourth and the fifth cases of the definition of cc':

- The third case specifies that the method constraint for the original method is replaced with the method constraint obtained from the method constraint in the method-constraint-wrap operation by replacing all the occurrences of the keyword `original` with the name, denoted by $\mathtt{m\$}\delta$ (where $\mathtt{m}$ is the name of the method that has to be wrapped and δ is the name of the delta module containing the method-constraint-wrap operation), of a new method with the original body of the method $\mathtt{m}$.
- The fourth case specifies that a method constraint for the method with name $\mathtt{m\$}\delta$, with the same expression constraints of the method constraint for the original method, is introduced.
- The fifth case specifies that, if a constraint for a method $\mathtt{m}$ is removed or replaced, then also the constraints $\mathtt{m\$}\cdots$ **with** $\cdots$ that might have been introduced by previously applied class-constraint operations are removed.

The following proposition states that the class-constraint operations application defined above indeed allows computing the class constraints for the class table APPLY(δ, CT) directly from the class-constraint operations for δ and the class constraints for CT. The proof is given in "Appendix C.1".

Proposition 2 *For every delta module* $\delta \in dom(\mathtt{DMT})$ *and for every class table* CT *such that* δ *is applicable to* CT, *if* $\vdash \mathtt{DMT}(\delta) : \mathscr{D}$ *and* $\vdash \mathtt{CT} : \mathscr{C}$, *then* $\vdash$ APPLY$(\delta, \mathtt{CT}) :$ CONSAPPLY$_\delta(\mathscr{D}, \mathscr{C})$.

For each valid feature configuration $\overline{\psi}$ of a strongly unambiguous product line, we write $\mathscr{C}_{\overline{\psi}}$ to denote the set of class constraints obtained by applying the sets of class-constraint operations inferred for the delta modules $\Delta(\overline{\psi})$ to the empty set of class constraints in any linear ordering that respects the total order on the partition of the delta modules specified in the product line declaration. The following corollary states that $\mathscr{C}_{\overline{\psi}}$ is indeed the set of class constraints of of the product $\mathtt{CT}_{\overline{\psi}}$.

Corollary 2 (of Proposition 2) *Let* L *be a strongly unambiguous IFΔJ SPL and* $\overline{\psi} \in \Phi$. *If the class signature table* $\mathtt{CST}_{\overline{\psi}}$ *is defined and* $\vdash \mathtt{delta}\ \delta\ \cdots : \mathscr{D}_\delta$ *(for all* $\delta \in \Delta(\overline{\psi})$*), then* $\vdash \mathtt{CT}_{\overline{\psi}} : \mathscr{C}_{\overline{\psi}}$.

Therefore, the class constraints of any product can be generated without generating the product.

Example 6 The application of the sets of class-constraint operations in Fig. 13 to the empty set of class constraints generates the class constraints given in Fig. 10.

7.3 Properties

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 states that the IFΔJ constraint-based type system is sound and complete with respect to the IFJ type system. The proof is given in "Appendix C.2".

Theorem 2 (Soundness and completeness of *IFJ* **constraint-based typing**) *Let* L *be a strongly unambiguous IFΔJ SPL and* $\overline{\psi} \in \Phi$.

(Soundness) If $\mathtt{CST}_{\overline{\psi}}$ *is defined and satisfies the sanity conditions for class signature tables,*[3] $\vdash \mathtt{delta}\ \delta\ \cdots : \mathscr{D}_\delta$ *for all* $\delta \in \Delta(\overline{\psi})$, *and* $\mathtt{CST}_{\overline{\psi}} \models$ FLAT$(\mathscr{C}_{\overline{\psi}})$, *then:*

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

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

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

[3] Cf. Sect. 4.2.

8 Enhancing early error recognition in delta modules

This section briefly discusses the issue of detecting as many type errors as possible in the code of the delta modules before performing the final product-based analysis phase that generates the type abstractions of the products (cf. Sect. 3).

In Sect. 6.2, we pointed out that the constraint-based typing rules for IFJ in Fig. 9 are able to recognize some ill-typed expressions during constraint generation. When these rules are used by the typing rules for IFΔJ in Fig. 9, it is safe to assume, without loss of generality, that no method with name `original` is defined. So, rule (CT- INVK) in Fig. 9 can be enhanced to detect more type errors when it is used to type the method-modify operations in the premise of rule (CT- S- WRAM) from Fig. 12. Namely, it can type calls to `this.original(`$\bar{\texttt{e}}$`)` by using for `original` the type of `m`, $\texttt{A}_1 \cdots \texttt{A}_p \rightarrow \texttt{C}$, instead of the type $\alpha_1 \cdots \alpha_n \rightarrow \beta$ (thus ensuring that `original` is called with the right number of parameters and avoiding to introduce the fresh variables $\alpha_1, \ldots, \alpha_n, \beta$).

The *family class signature* `FCS` of a class `C` maps the keyword `extends` to a non-empty set of class names and maps field/method names to a non-empty finite set of types such that:

- `FCS(extends)` contains a given class `D` if and only if the `D` is the direct superclass of `C` in some valid product, and
- for each field/method name `a`, the set `FCS(a)` contains a given type if and only if the field/method `a` is defined with that type in the class `C` in some valid product.

The *family class signature table* `FCST` of a delta-oriented product line is a mapping from class names to family class signatures. It can be straightforwardly generated by composing the signatures of all delta modules of the product line by ignoring the `removes` operations. Many type errors could be detected by checking the set of class-constraint operations inferred for a delta module against the family class signature table of the product line. If in a table each set in the domain of the family class signature is a singleton and the subclassing relation is acyclic, then the only type errors in the products of the product line that cannot be detected by performing these checks are those due to fact that in some product some required field, method, class or subclass is missing. A smart implementation of the constraint-based type system could annotate each generated constraint with the location of the associated code in delta module. This would make it possible to trace the source of typing errors that are detected when checking constraints back to the originating delta modules. These checks should be able to provide for delta modules the same guarantees provided for feature modules by the lightweight global consistency checks illustrated in [48].

9 Related work

Delta-oriented programming [42,44] is an extension of feature-oriented programming [6]. In [42], the general ideas of DOP are presented and compared conceptually and empirically to FOP. The presentation in [42] uses the notion of a core product, that is a designated product of the product line and the starting point of product generation. In [44], the notion of the core product is dropped so that product generation only relies on delta modules. This makes DOP even more flexible to support proactive, extractive and reactive product line engineering [34]. Proactive product line engineering aims at developing the product line from scratch, extractive product line engineering turns existing products into a product line, and proactive product line engineering stepwise evolves an initial set of product variants. Furthermore, DOP without the notion of a core product [44] allows a direct embedding of FOP into DOP. In this

paper, we use the notion of DOP as presented in [44] in order to provide a compositional approach for type checking delta-oriented product lines of JAVA programs.

DOP and FOP are compositional approaches [30] for implementing SPLs in which code fragments are associated with product features and assembled to implement a particular feature configuration. Other compositional approaches use aspects [4,29], mixins [45], hyperslices [47] or traits [7,20] to implement product line variability (see [36] for a discussion of some of them with respect to FOP).

Feature-oriented Programming Various approaches to ensure the type safety of feature-oriented product lines can be found in literature [2,3,19,35,48]. The type system of LIGHTWEIGHT FEATURE JAVA (LFJ) [19] is the closest to our proposal. The calculus LFJ, based on LJ (LIGHTWEIGHT JAVA) [46], provides a formalization of FOP as implemented in AHEAD [6], together with a constraint-based type system that supports a feature-based phase and a final family-based phase (cf. Sect. 3). The approach consists of three technical concepts:

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 system 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 a product-based analysis phase relying directly on the constraints for the feature modules (as done in the present paper) requires an explicit iteration on the valid feature configurations. Checking the satisfiability of the propositional formula 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 [19]. If the formula is not satisfiable, it is not straightforward to trace the error back to the feature module that causes the error.

Due to the additional flexibility provided by DOP, there is currently no analogue of the third concept of FOP type-checking available for DOP type-checking. 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.

The `original` construct of IFΔJ is similar to the `Super` construct of AHEAD. LFJ formalizes a simplified version of the `Super` construct. Namely, a call `Super()` represents a call to the unmodified method where the formal parameters of the modified method are passed implicitly as arguments and the body of the modified method is built by replacing the occurrence of `Super()` with the body of the original method.

The LFJ and the IFΔJ type systems have similarities with a type system proposed in [1] to type-check, compile and link code fragments. A code fragment is formalized as a set of

JAVA classes. Similarly to feature/delta modules, code fragments can reference definitions provided in other code fragments. The purpose of the type system presented in [1] is to ensure 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 key idea is to define a polymorphic form of bytecode containing type variables (ranging over class names) and equipped with a set of constraints involving type variables. Polymorphic bytecode, which is generated by compiling each code fragment in isolation, provides a representation for all the (standard) bytecode that can be obtained by replacing type variables with classes satisfying the associated constraints. During the linking phase, constraints are solved causing the instantiation of the type variables (thus transforming polymorphic bytecode into standard bytecode.)

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) [3]. 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 (each feature module is analyzed during a family-based phase by relying on information of the complete product line).

Aspect-oriented Programming Both delta-oriented and aspect-oriented programming (AOP) [33] 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 [32], 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 run-time 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 also allow the programmer to intercept most execution statements in a program, 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.

Another difference between AOP and DOP is that in DOP, a set of delta modules and a product line declaration defines a set of products. On the contrary, in AOP, 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. In order to use AOP to implement SPLs it might be useful to provide a product-line declaration with **when** clauses for aspects.

The *A* calculus [25] 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., [13,18]), 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, thus `original` can be used to implement a wrapper method with the original signature. The `original` construct is, thus, typed with the same type as the return type of the original method (cf. Sect. 5).

Class Composition Mechanisms as a Language Construct The language GBETA [21,22] provides a mechanism for combining mixin-based classes and methods with propagation (i.e., combining classes and methods can imply an implicit propagation of class or method combinations). The whole approach is statically typed. The language supports inheritance between *virtual classes* [37], allowing to implement higher-order hierarchies [23]. These mechanisms are used in [24] to present a solution of the expression problem [50] which consists of classes and methods to be added to a given class family.

In [51] mixin class composition mechanisms of SCALA [38] are used to show other solutions to the expression problem. The mixin class composition in SCALA borrows both from the mixin construct presented in [11] and the trait composition mechanism presented in [20]. Since SCALA does not provide a direct language mechanism to perform deep mixin composition (which basically corresponds to GBETA's propagation illustrated above), some more code is required for solving some parts of the expression problem compared to the approach in GBETA [24].

The "meta" and "generative" flavor of IFΔJ shares with GBETA's higher-order hierarchies [23] that class hierarchies are not modified (as in AOP): different separate copies are created. However, although in GBETA a new root in a class hierarchy as well as a new intermediate class can be introduced, still the inheritance relations must be kept. Thus, it is not possible to completely change the superclass of an existing class, like in IFΔJ with the `extending` clause. Moreover, in IFΔJ, besides extending existing classes, it is also possible to remove parts of the classes, e.g., removing methods and fields.

The main differences between IFΔJ and the class composition mechanisms above is that the former provides two levels in the language: one for specifying the execution parts

of the program, and one for manipulating and composing the code blocks to build new products.

HYPER/J [39] implements multi-dimensional separation of concerns [47] for JAVA. Once the concerns of an application are identified, HYPER/J provides mechanisms to specify modules (*hyperslices*) in terms of those concerns, and to synthesize components by integrating those modules. Starting from standard JAVA class files, it produces new JAVA class files. Specifications of concerns and the relationships among them to be used for the actual compositions are provided in a control file. The declarative completeness requirement of hyperslices (i.e., an hyperslice declares everything to which it refers) is intended to decouple hyperslices from each other. These explicit declarations might become a burden for the programmer; in IFΔJ the constraints are instead inferred by the type system and this should make delta modules easier to reuse and should fit better the context of SPL development.

The design of the constraint-based type system for IFΔJ involves issues similar to those considered in type-checking of dynamic classes [27]. 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, allowing more flexible removals.

10 Conclusions and future work

We have provided a foundation for compositional type checking of delta-oriented product lines that supports a feature-based analysis phase and a final product-based analysis phase by relying on an abstraction of product generation. During the feature-based analysis phase each delta module is analyzed in in isolation such that the analysis results can be re-used across different product lines.

The initial ideas of DOP presented in [42] (see Sect. 9 for a comparison with the approach of this paper) have been implemented as an Eclipse IDE and a standalone compiler, which can be found at http://deltaj.sourceforge.net. At the same site, a new implementation of DOP based on the approach presented in this paper, is also available. This implementation is still in a development stage, though it implements most of the features of IFΔJ. In particular, besides the language of delta modules and product-line declarations, we also have a DSL for the constraints: the IFΔJ compiler, in addition to generating Java code for the products, also generates the textual constraints; this way we can easily show to the programmer possible errors due to configurations which violate constraints. All these features are integrated in Eclipse.

In future work, we would like to investigate the possibility of enhancing the proposed type checking mechanism for DOP with the third concept of the type checking approach pursued for LFJ (cf. Sect. 9). Another interesting future research direction is to extend DOP to express the change of the feature configuration of a product at run-time [40,41]. A first attempt in this direction has been presented in [16] with the formal foundations being laid out in [15]. To achieve this goal, it might be useful to equip IFΔJ with a direct semantics as done for FFJ$_{PL}$ [2].

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 following the ideas behind FEATUREHOUSE. A starting point is the trait-based prototypical programming language TRAITRECORDJ [7,8] (see also [9,10]). In TRAITRECORDJ, classes are assembled from interfaces, records (providing fields) and traits [20] (providing methods) that can be directly manipulated by designated composition operations. These operations make TRAITRECORDJ a good candidate for enhancing the flexibility of delta modules and providing further support for code reuse.

Acknowledgments We are grateful to the anonymous referees of Acta Informatica for insightful comments, suggestions for improving the presentation and pointers to related work. We also thank Luca Padovani and Shmuel Tyszberowicz for useful comments on a preliminary version of this paper.

Appendix A: IFJ reduction and type soundness

The length of a sequence $\bar{\mathtt{e}}$ is denoted by $\#(\bar{\mathtt{e}})$.

A.1 IFJ reduction

In order to properly model imperative features of IFJ, we introduce the concepts of *address* and *heap*. *Addresses*, ranged over by the metavariable ι, are the elements of the denumerable set $\mathbf{I}$. *Values*, ranged over by the metavariable $\mathtt{v}$ are either addresses or `null`. *Objects* are denoted by $\langle \mathtt{C}, \bar{\mathtt{f}} = \bar{\mathtt{v}} \rangle$, where $\mathtt{C}$ is the class of the object, $\bar{\mathtt{f}}$ are the name of the fields and $\bar{\mathtt{v}}$ are the values of the fields. A heap $\mathcal{H}$ is a mapping from *addresses* to *objects*. The empty heap will be denoted by $\emptyset$. *Runtime expressions* are obtained from expressions by replacing all the variables (including `this`) by addresses. We will use e to denote runtime expressions.

The states of a computation are represented by means of configurations. A *configuration* is a pair consisting of a heap and a runtime expression, written $\mathcal{H}, e$. The reduction relation has the form $\mathcal{H}, e \longrightarrow \mathcal{H}', e'$, to read "the configuration $\mathcal{H}, e$ reduces to the configuration $\mathcal{H}', e'$ in one step". The initial configuration associated to a program $\mathtt{CT}$ is $\emptyset, \mathtt{e}$ (where $\mathtt{e}$ is the body of method `main` of class `Main`).

The reduction rules shown in Fig. 14, using the standard notions of *computation rules* and *congruence rules*, ensure that the computation is carried on according to a call-by-value reduction strategy.

The operational semantics uses the auxiliary functions *mbody* and *fields*, which are defined in Fig. 15.

A.2 IFJ type soundness

In order to be able to formulate the type soundness of IFJ as a subject reduction theorem and a progress theorem for the small-step semantics, we need to formulate a type system for runtime expressions. Expressions containing either a *stupid cast* (a notion introduced in [26]), i.e., a cast where the subject and the target are unrelated, or a *stupid selection*, i.e., a field selection `null.f` or a method invocation $\mathtt{null.m}(\cdots)$, are not well typed according to the IFJ (source level) type system. However, a runtime expression without stupid casts and stupid selections may reduce to a runtime expression containing either a stupid cast or a stupid

Computation rules:

$$\frac{\mathscr{H}(\iota) = \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{v}} \rangle}{\mathscr{H}, \iota.\mathtt{f}_i \longrightarrow \mathscr{H}, \mathtt{v}_i} \quad \text{(R-FIELD)}$$

$$\frac{\mathscr{H}(\iota) = \langle \mathtt{C}, \ldots \rangle \qquad \mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}_0)}{\mathscr{H}, \iota.\mathtt{m}(\overline{\mathtt{v}}) \longrightarrow \mathscr{H}, [\overline{\mathtt{x}} \leftarrow \overline{\mathtt{v}}, \mathtt{this} \leftarrow \iota]\mathtt{e}_0} \quad \text{(R-INVK)}$$

$$\frac{\iota \notin \mathit{dom}(\mathscr{H}) \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\mathscr{H}, \mathtt{new}\ \mathtt{C}() \longrightarrow \mathscr{H} \cup \{\iota \mapsto \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{null}} \rangle\}, \iota} \quad \text{(R-NEW)}$$

$$\frac{\mathscr{H}(\iota) = \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{v}} \rangle}{\mathscr{H}, \iota.\mathtt{f}_i = \mathtt{v} \longrightarrow \mathscr{H}[\iota \mapsto \langle \mathtt{C}, \ldots, \mathtt{f}_i = \mathtt{v}, \ldots \rangle], \mathtt{v}} \quad \text{(R-ASSIGN)}$$

$$\frac{\mathscr{H}(\iota) = \langle \mathtt{C}, \overline{\mathtt{f}} = \overline{\mathtt{v}} \rangle \qquad \mathtt{C} <: \mathtt{D}}{\mathscr{H}, (\mathtt{D})\ \iota \longrightarrow \mathscr{H}, \iota} \quad \text{(R-CAST)}$$

$$\mathscr{H}, (\mathtt{D})\ \mathtt{null} \longrightarrow \mathscr{H}, \mathtt{null} \quad \text{(R-CASTN)}$$

Congruence rules:

$$\frac{\mathscr{H}, e \longrightarrow \mathscr{H}', e'}{\mathscr{H}, e.\mathtt{f} \longrightarrow \mathscr{H}', e'.\mathtt{f}} \qquad \frac{\mathscr{H}, e \longrightarrow \mathscr{H}', e'}{\mathscr{H}, e.\mathtt{m}(\overline{e}) \longrightarrow \mathscr{H}', e'.\mathtt{m}(\overline{e})}$$

$$\frac{\mathscr{H}, e_i \longrightarrow \mathscr{H}', e_i'}{\mathscr{H}, \mathtt{v}.\mathtt{m}(\overline{\mathtt{v}}, e_i, \overline{e}) \longrightarrow \mathscr{H}', \mathtt{v}.\mathtt{m}(\overline{\mathtt{v}}, e_i', \overline{e})} \qquad \frac{\mathscr{H}, e \longrightarrow \mathscr{H}', e'}{\mathscr{H}, e.\mathtt{f} = e_0 \longrightarrow \mathscr{H}', e'.\mathtt{f} = e_0}$$

$$\frac{\mathscr{H}, e \longrightarrow \mathscr{H}', e'}{\mathscr{H}, \mathtt{v}.\mathtt{f} = e \longrightarrow \mathscr{H}', \mathtt{v}.\mathtt{f} = e'} \qquad \frac{\mathscr{H}, e \longrightarrow \mathscr{H}', e'}{\mathscr{H}, (\mathtt{C})\ e \longrightarrow \mathscr{H}', (\mathtt{C})\ e'}$$

Fig. 14 IFJ: operational semantics

Fig. 15 IFJ: auxiliary functions

Field lookup:

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

$$\frac{\mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \overline{\mathtt{MD}}\ \} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}}}$$

Method body lookup:

$$\frac{\mathit{aDef}(\mathtt{C})(\mathtt{m}) = \mathtt{B}\ \mathtt{m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e})}$$

selection. The type system for runtime expressions contains a rule for typing stupid casts, and a rule for assigning any type T to the value null (so that stupid selection can be typed).

Typing rules for runtime expressions are shown in Fig. 16; these rules use the environment Σ, which is a finite (possibly empty) mapping from addresses to class names, and they are

Runtime expression typing $\boxed{\Sigma \vdash' \boldsymbol{e} : \mathtt{T}}$

$$\Sigma \vdash' \iota : \Sigma(\iota) \qquad \text{(RT-Addr)}$$

$$\frac{\Sigma \vdash' e : \mathtt{C} \qquad \mathit{aType}(\mathtt{C})(\mathtt{f}) = \mathtt{A}}{\Sigma \vdash' e.\mathtt{f} : \mathtt{A}} \qquad \text{(RT-Field)}$$

$$\frac{\Sigma \vdash' e : \mathtt{C} \qquad \mathit{aType}(\mathtt{C})(\mathtt{m}) = \bar{\mathtt{A}} \to \mathtt{B} \qquad \Sigma \vdash' \bar{e} : \bar{\mathtt{T}} \qquad \bar{\mathtt{T}} <: \bar{\mathtt{A}}}{\Sigma \vdash' e.\mathtt{m}(\bar{e}) : \mathtt{B}} \qquad \text{(RT-Invk)}$$

$$\frac{\mathtt{C} \in \mathit{dom}(\mathtt{CST})}{\Sigma \vdash' \mathtt{new\ C}() : \mathtt{C}} \qquad \text{(RT-New)}$$

$$\frac{\Sigma \vdash' e : \mathtt{T} \qquad \mathtt{T} <: \mathtt{C}}{\Sigma \vdash' (\mathtt{C})\, e : \mathtt{C}} \qquad \text{(RT-UCast)}$$

$$\frac{\Sigma \vdash' e : \mathtt{B} \qquad \mathtt{C} <: \mathtt{B} \qquad \mathtt{C} \neq \mathtt{B}}{\Sigma \vdash' (\mathtt{C})\, e : \mathtt{C}} \qquad \text{(RT-DCast)}$$

$$\frac{\Sigma \vdash' e : \mathtt{T} \qquad \mathtt{C} \not<: \mathtt{T} \qquad \mathtt{T} \not<: \mathtt{C} \qquad \textit{stupid warning}}{\Sigma \vdash' (\mathtt{C})\, e : \mathtt{C}} \qquad \text{(RT-SCast)}$$

$$\Sigma \vdash' \mathtt{null} : \mathtt{T} \qquad \mathtt{T} \in \{\bot\} \cup \{\mathtt{Object}\} \cup \mathit{dom}(\mathtt{CST}) \qquad \text{(RT-Null)}$$

$$\frac{\Sigma \vdash' e_0.\mathtt{f} : \mathtt{C} \qquad \Sigma \vdash' e_1 : \mathtt{T} \qquad \mathtt{T} <: \mathtt{C}}{\Sigma \vdash' e_0.\mathtt{f} = e_1 : \mathtt{C}} \qquad \text{(RT-Assign)}$$

Well-formed heap $\boxed{\Sigma \Vdash \mathscr{H}}$

$$\frac{\begin{array}{l}\mathit{dom}(\mathscr{H}) = \mathit{dom}(\Sigma) \\ \forall \iota \in \mathit{dom}(\mathscr{H}),\end{array} \quad \mathscr{H}(\iota) = \langle \mathtt{C}, \mathtt{f}_1 = \mathtt{v}_1, \ldots, \mathtt{f}_n = \mathtt{v}_n \rangle \quad \textbf{implies} \quad \left(\begin{array}{l}\Sigma(\iota) = \mathtt{C} \\ \mathit{fields}(\mathtt{C}) = \mathtt{C}_1\ \mathtt{f}_1, \ldots, \mathtt{C}_n\ \mathtt{f}_n \\ \forall i \in 1..n, \quad \Sigma \vdash' \mathtt{v}_i : \mathtt{T}_i \quad \mathtt{T}_i <: \mathtt{C}_i\end{array}\right)}{\Sigma \Vdash \mathscr{H}} \qquad \text{(WF-Heap)}$$

Well-typed configuration $\boxed{\Sigma \vdash' \mathscr{H}, \boldsymbol{e} : \mathtt{T}}$

$$\frac{\Sigma \vdash' e : \mathtt{T} \qquad \Sigma \Vdash \mathscr{H}}{\Sigma \vdash' \mathscr{H}, e : \mathtt{T}} \qquad \text{(WF-Conf)}$$

Fig. 16 IFJ: typing rules for runtime expressions and heaps

of the form $\Sigma \vdash' e : \mathtt{T}$. In Fig. 16 we also present the notion of *well-formed heap* and of *well-formed configuration*. The notion of well-formed heap ensures that the environment Σ maps all the addresses in the heap into the type of the corresponding object and that for every object stored in the heap, the fields of the object contain appropriate values.

Type soundness can be proved by using the standard technique of subject reduction and progress theorems.

Lemma 1 *If a* $\mathit{aType}\mathtt{C_0m} = \bar{\mathtt{D}} \to \mathtt{D}$ *and* $\mathit{mbody}(\mathtt{C_0}, \mathtt{m}) = (\bar{\mathtt{x}}, \mathtt{e})$ *then for some* $\mathtt{D_0}$ *and some* $\mathtt{T} <: \mathtt{D}$ *we have* $\mathtt{C_0} <: \mathtt{D_0}$ *and* $\mathtt{this} : \mathtt{D_0},\ \bar{\mathtt{x}} : \bar{\mathtt{D}} \vdash \mathtt{e} : \mathtt{T}$.

Proof By straightforward induction on the derivation of $mbody(\mathtt{C}_0, \mathtt{m})$, that is, on $aDef(\mathtt{C}_0)(\mathtt{m})$.

Lemma 2 (Substitution) *If*

1. $\Sigma \vdash' \iota.\mathtt{m}(\overline{\mathtt{v}}) : \mathtt{D}$ *where* $\Sigma(\iota) = \mathtt{C}_0$ *for some* Σ, $\mathtt{C}_0$ *and* $\mathtt{D}$,
2. $aType(\mathtt{C}_0)(\mathtt{m}) = \bar{\mathtt{A}} \to \mathtt{D}$, *and*
3. $mbody(\mathtt{C}_0, \mathtt{m}) = (\bar{\mathtt{x}}, \mathtt{e})$,

then for some $\mathtt{C}' <: \mathtt{D}$ *we have* $\Sigma \vdash' [\bar{\mathtt{x}} \leftarrow \overline{\mathtt{v}}, \mathtt{this} \leftarrow \iota]\mathtt{e} : \mathtt{C}'$.

Proof By hypothesis 1. and 2. and by Lemma 1, for some $\mathtt{C}$ and some $\mathtt{T} <: \mathtt{D}$, we have $\mathtt{C}_0 <: \mathtt{C}$ and $\mathtt{this} : \mathtt{C}, \bar{\mathtt{x}} : \bar{\mathtt{A}} \vdash \mathtt{e} : \mathtt{T}$.

The proof then proceeds by structural induction on the derivation of $\mathtt{this} : \mathtt{C}, \bar{\mathtt{x}} : \bar{\mathtt{A}} \vdash \mathtt{e} : \mathtt{T}$. We present only a few interesting cases (the cases for casts are the same as in FJ, in particular, for (T- DCast) we can use (RT- SCast)). Note that, by rule (RT- Invk), $\Sigma \vdash' \overline{\mathtt{v}} : \bar{\mathtt{C}}$ for some $\bar{\mathtt{C}}$ such that $\bar{\mathtt{C}} <: \bar{\mathtt{A}}$ (in particular, $\mathtt{C}_i = \mathtt{A}_i$ when $\mathtt{v}_i = \mathtt{null}$ by rule (RT- Null)).

Case (T- Var) In this case $\mathtt{e} = \mathtt{x}_i$ for some $\mathtt{x}_i \in \bar{\mathtt{x}}$; $[\bar{\mathtt{x}} \leftarrow \overline{\mathtt{v}}, \mathtt{this} \leftarrow \iota]\mathtt{x}_i = \mathtt{v}_i$ and $\Sigma \vdash' \mathtt{v}_i : \mathtt{C}_i$ for some $\mathtt{C}_i$ such that $\mathtt{C}_i <: \mathtt{A}_i$; letting $\mathtt{C}_i = \mathtt{C}'$ finishes the case.

Case (T- Field) In this case $\mathtt{e} = \mathtt{e}'.\mathtt{f}$. By rule (T- Field) we have $\mathtt{this} : \mathtt{C}, \bar{\mathtt{x}} : \bar{\mathtt{A}} \vdash \mathtt{e}' : \mathtt{C}'$ and $aType(\mathtt{C}')(\mathtt{f}) = \mathtt{A}$. By the induction hypothesis, $\Sigma \vdash' [\bar{\mathtt{x}} \leftarrow \overline{\mathtt{v}}, \mathtt{this} \leftarrow \iota]\mathtt{e}' : \mathtt{C}''$ for some $\mathtt{C}'' <: \mathtt{C}'$. The thesis follows from $aType(\mathtt{C}'')(\mathtt{f}) = aType(\mathtt{C}')(\mathtt{f}) = \mathtt{A}$.

Case (T- Invk) In this case $\mathtt{e} = \mathtt{e}'.\mathtt{m}(\bar{\mathtt{e}})$. Similar to the previous case, using the induction hypothesis on $\mathtt{e}'$ and $\bar{\mathtt{e}}$, and using the fact that $aType(\mathtt{C}'')(\mathtt{m}) = aType(\mathtt{C}')(\mathtt{m})$ if $\mathtt{C}'' <: \mathtt{C}'$.

Case (T- Assig) In this case $\mathtt{e}$ is of the form $\mathtt{e}_0.\mathtt{f} = \mathtt{e}_1$. By (T- Assig) we have $\mathtt{this} : \mathtt{C}, \bar{\mathtt{x}} : \bar{\mathtt{A}} \vdash \mathtt{e}_0.\mathtt{f} : \mathtt{A}, \mathtt{this} : \mathtt{C}, \bar{\mathtt{x}} : \bar{\mathtt{A}} \vdash \mathtt{e}_1 : \mathtt{T}_1$ for some $\mathtt{T}_1 <: \mathtt{A}$. The thesis follows from the induction hypothesis and the transitivity of $<:$.

Lemma 3 (Weakening) *If* $\Sigma \vdash' e : \mathtt{T}$ *then* $\Sigma, \iota : \mathtt{C} \vdash' e : \mathtt{T}$.

Proof Straightforward induction on the derivation of $\Sigma \vdash' e : \mathtt{T}$.

Theorem 3 (Subject reduction) *If* $\Sigma \Vdash \mathscr{H}$, $\Sigma \vdash' e : \mathtt{T}$ *and* $\mathscr{H}, e \longrightarrow \mathscr{H}', e'$ *then there exists* $\Sigma' \supseteq \Sigma$ *such that* $\Sigma' \Vdash \mathscr{H}'$, $\Sigma' \vdash' e' : \mathtt{T}'$ *for some* $\mathtt{T}' <: \mathtt{T}$.

Proof The proof is by induction on a derivation of $\mathscr{H}, e \longrightarrow \mathscr{H}', e'$, with a case analysis on the reduction rule used. We show only the most interesting cases for computation rules; for congruence rules simply use the induction hypothesis (using Lemma 3).

Case (R- Field) The last applied rule is $\mathscr{H}, \iota.\mathtt{f}_i \longrightarrow \mathscr{H}, \mathtt{v}_i$ where $\mathscr{H}(\iota) = \langle \mathtt{C}, \bar{\mathtt{f}} = \overline{\mathtt{v}} \rangle$. By hypothesis $\Sigma \vdash' \iota.\mathtt{f}_i : \mathtt{T}_i$. and by (WF- Heap) we have $\Sigma \vdash' \mathtt{v}_i : \mathtt{T}'_i$ for some $\mathtt{T}'_i <: \mathtt{C}_i$. Thus we have the thesis.

Case (R- Invk) The last applied rule is $\frac{\mathscr{H}(\iota)=\langle \mathtt{C}, \bar{\mathtt{f}}=\overline{\mathtt{v}} \rangle \; mbody(\mathtt{m},\mathtt{C})=(\bar{\mathtt{x}},\mathtt{e}_0)}{\mathscr{H}, \iota.\mathtt{m}(\overline{\mathtt{v}}) \longrightarrow \mathscr{H}, [\bar{\mathtt{x}} \leftarrow \overline{\mathtt{v}}, \mathtt{this} \leftarrow \iota]\mathtt{e}_0}$

By hypothesis $\Sigma \vdash' \iota.\mathtt{m}(\overline{\mathtt{v}}) : \mathtt{T}$. Since the last applied typing rule must be (RT- Invk), we have $\mathtt{T} = \mathtt{B}$ for some $\mathtt{B}$. Then the thesis follows by applying Lemma 2.

Case (R- New) Let $\Sigma' = \Sigma \cup \{\iota : \mathtt{C}\}$. By hypothesis $\Sigma \Vdash \mathscr{H}$, and by applying (WF- Heap) we also have $\Sigma' \Vdash \mathscr{H} \cup \{\iota \mapsto \mathtt{C}, \bar{\mathtt{f}} = \mathtt{null}\}$. $\Sigma' \vdash' \iota : \mathtt{C}$ follows from (RT- Addr).

Case (R- Assign) By rule (RT- Assign) we have that $\Sigma \vdash' \mathtt{v} : \mathtt{T}'$ and $\mathtt{T}' <: \mathtt{T}$ for some $\mathtt{T}'$. By hypothesis $\Sigma \Vdash \mathscr{H}$, and by applying (WF- Heap) we also have $\Sigma \Vdash \mathscr{H}[\mathscr{H}(\iota) \mapsto \langle \mathtt{C}, \ldots, \mathtt{f}_i = \mathtt{v}, \ldots \rangle]$.

Lemma 4 *Let $\mathscr{H}$, e be a well-typed configuration.*

1. *If $e = \iota.\mathtt{f}$ then $\mathscr{H}(\iota) = \langle \mathtt{C}, \cdots \rangle$ with $aType(\mathtt{C})(\mathtt{f}) = \mathtt{A}$ for some class name $\mathtt{A}$.*
2. *If $e = \iota.\mathtt{m}(\bar{e})$ then $\mathscr{H}(\iota) = \langle \mathtt{C}, \cdots \rangle$ with $aType(\mathtt{C})(m) = \bar{\mathtt{A}} \to \mathtt{B}$ and $\sharp(\bar{\mathtt{A}}) = \sharp(\bar{e})$.*

Proof Straightforward.

In order to formulate in a compact way the statement of the progress theorem we introduce the notion of evaluation context for IFJ runtime expressions. The set of evaluation context for IFJ runtime expressions is defined as follows:

$$E \;::=\; [\,] \;\mid\; E.\mathtt{f} \;\mid\; E.\mathtt{m}(\bar{\mathtt{e}}) \;\mid\; \mathtt{v}.\mathtt{m}(\bar{\mathtt{v}}, E.\bar{\mathtt{e}}) \;\mid\; (\mathtt{C})E \;\mid\; \mathtt{v}.\mathtt{f} = E$$

Theorem 4 (Progress) *Let $\mathscr{H}$, e be a well-typed normal form. Then*

1. *either e is a value, or*
2. *for some evaluation context E we can express e as*
   (a) *either $E\,[(\mathtt{A})\iota]$ such that $\mathscr{H}(\iota) = \langle \mathtt{B}, \cdots \rangle$ with $\mathtt{B} \not<: \mathtt{A}$, or*
   (b) *$E\,[\mathtt{null.f}]$ for some $\mathtt{f}$, or*
   (c) *$E\,[\mathtt{null.m}(\bar{\mathtt{v}})]$ for some $\mathtt{m}$ and $\bar{\mathtt{v}}$, or*
   (d) *$E\,[\mathtt{null.f} = \mathtt{v}]$ for some $\mathtt{f}$ and $\mathtt{v}$.*

Proof Straightforward induction on typing derivations using Lemma 4.

Lemma 5 *If $\bullet \vdash \mathtt{e} : \mathtt{T}$ then $\bullet \vdash' \mathtt{e} : \mathtt{T}$.*

Proof Straightforward induction on typing derivations.

Theorem 5 (Type Soundness) *If $\vdash$ CT OK, $\mathtt{CT}(\mathtt{Main}) = \mathtt{class\ Main\ \{\ C\ main()\ \{\ return\ e;\ \}\ \}}$, $\bullet \vdash \mathtt{e} : \mathtt{T}$ and $\emptyset, e \longrightarrow^{\star} \mathscr{H}, \mathtt{e}'$ with $\mathscr{H}, e'$ a normal form. Then e' is*

1. *either $\mathtt{null}$,*
2. *or an address ι such that $\mathscr{H}(\iota) = \langle \mathtt{C}, \cdots \rangle$ with $\mathtt{C} <: \mathtt{T}$,*
3. *or an expression containing $(\mathtt{A})\iota$ such that $\mathscr{H}(\iota) = \langle \mathtt{B}, \cdots \rangle$ with $\mathtt{B} \not<: \mathtt{A}$,*
4. *or an expression containing either $\mathtt{null.f}$ or $\mathtt{null.m}(\bar{\mathtt{v}})$ for some $\mathtt{f}$, $\mathtt{m}$ and $\bar{\mathtt{v}}$.*

Proof Follows from Lemma 5, Theorems 3 and 4.

Appendix B: Soundness and completeness of IFJ constraint-based typing

Lemma 6 *Let CT be a IFJ program, $\mathtt{C} \in dom(\mathtt{CT})$, $\mathtt{m} \in dom(\mathtt{C})$, $\mathtt{CST} = signature(\mathtt{CT})$, CST satisfy the sanity conditions for class signature tables, $\mathtt{CT}(\mathtt{C})(\mathtt{m}) = \mathtt{B\ m\ (\bar{A}\ \bar{x})\{return\ e';\}}$ and e be a subexpression of $\mathtt{e}'$.*

(Soundness) Let $\mathtt{this} : \mathtt{C}, \bar{\mathtt{x}} : \bar{\mathtt{A}} \vdash \mathtt{e} : \tau \mid \mathscr{E}$ and $\mathtt{CST} \models \mathscr{E} \Rightarrow \mathbf{s}$. Then $\mathtt{this} : \mathtt{C}, \bar{\mathtt{x}} : \bar{\mathtt{A}} \vdash \mathtt{e} : \mathbf{s}(\tau)$.
(Completeness) Let $\mathtt{this} : \mathtt{C}, \bar{\mathtt{x}} : \bar{\mathtt{A}} \vdash \mathtt{e} : \mathtt{T}$. Then there exist τ, $\mathscr{E}$ and $\mathbf{s}$ such that: $\mathtt{this} : \mathtt{C}, \bar{\mathtt{x}} : \bar{\mathtt{A}} \vdash \mathtt{e} : \tau \mid \mathscr{E}$, $\mathtt{CST} \models \mathscr{E} \Rightarrow \mathbf{s}$ and $\mathbf{s}(\tau) = \mathtt{T}$.

Proof (Soundness) By structural induction on the derivations in the constraint-based type system for expressions in Fig. 9, exploiting the rules in Fig. 8.

The cases (CT- VAR), (CT- NULL) and (CT- NEW) are immediate by rules (T- VAR), (T- NULL) and (T- NEW), respectively. For rule (CT- NEW) observe that $\mathtt{CST} \models \{\mathbf{c}lass(\mathtt{C})\}$ implies $\mathtt{C} \in dom(\mathtt{CST})$.

Case (CT- INVK). Assume $\mathtt{this} : \mathtt{C}, \mathtt{x}_1 : \mathtt{A}_1, \ldots, \mathtt{x}_p : \mathtt{A}_p \vdash \mathtt{e}_0.\mathtt{m}(\mathtt{e}_1, ..., \mathtt{e}_n) : \beta \mid (\cup_{i\in\{0,...,n\}}\mathscr{E}_i) \cup \mathscr{E}$ and $\mathtt{CST} \models \mathscr{E} \cup (\cup_{i\in\{0,...,n\}}\mathscr{E}_i) \Rightarrow \mathbf{s}$. From the premises of rule (CT- INVK) we have that:

- $\Gamma \vdash \mathtt{e}_0 : \eta \mid \mathscr{E}_0$,
- $\Gamma \vdash \mathtt{e}_i : \tau_i \mid \mathscr{E}_i \ ^{(i\in 1..n)}$,
- $\alpha_1, ..., \alpha_n, \beta$ fresh, and
- $\mathscr{E} = \{\mathbf{meth}(\eta, \mathtt{m}, \alpha_1 \cdots \alpha_n \to \beta), \mathbf{subtype}(\tau_1\alpha_1), \ldots, \mathbf{subtype}(\tau_n\alpha_n)\}$.

Assume (without loss of generality) that for all $i \neq j \in 0..n$ the set of the type variables in $\{\tau_i\} \cup \mathscr{E}_i$ and the set of the type variables in $\{\tau_j\} \cup \mathscr{E}_j$ (with $\tau_0 = \eta$) are pairwise disjoint. From the rules in Fig. 8 we have that $\mathbf{s} = \mathbf{s}' \circ \mathbf{s}_n \circ \cdots \circ \mathbf{s}_0$ for some $\mathbf{s}', \mathbf{s}_n, \ldots, \mathbf{s}_0$ such that:

- $\mathtt{CST} \models \mathscr{E}_0 \Rightarrow \mathbf{s}_0$ and $\mathbf{s}_0(\eta) = \mathtt{C}_0$, for some $\mathtt{C}_0$,
- (for all $i \in 1..n$) $\mathtt{CST} \models \mathscr{E}_i \Rightarrow \mathbf{s}_i$ and $\mathbf{s}_i(\tau_i) = \mathtt{T}_i$, for some $\mathtt{T}_i$,
- $\mathtt{CST} \models \mathbf{s}_n \circ \cdots \circ \mathbf{s}_0(\mathscr{E}) \Rightarrow \mathbf{s}'$ where $\mathbf{s}' = [\mathtt{BA}_1 \cdots \mathtt{A}_n/\beta\alpha_1 \cdots \alpha_n]$, $aType(\mathtt{C}_0)(\mathtt{m}) = \mathtt{A}_1 \cdots \mathtt{A}_n \to \mathtt{B}$ and $\mathtt{T}_i <: \mathtt{A}_i \ ^{(i\in 1..n)}$.

By induction we have that:

- $\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0$,
- $\Gamma \vdash \mathtt{e}_i : \mathtt{T}_i \ ^{(i\in 1..n)}$.

Then, by rule (T- INVK), we get $\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \mathtt{B}$.
The remaing cases (T- FIELD), (T- INVK), (T- NEW), (T- ASSIG),(T- CAST) are similar to the previous case.
(Completeness) By structural induction on the derivations in the type system for expressions in Fig. 4, exploiting the rules in Fig. 8.

- The cases (T- VAR), (T- NULL) and (T- NEW) are immediate by rules (CT- VAR), (CT- NULL) and (T- NEW), respectively. For rule (T- NEW) observe that $\mathtt{C} \in dom(\mathtt{CST})$ implies $\mathtt{CST} \models \{\mathbf{c}lass(\mathtt{C})\}$.

Case (T- INVK). Assume $\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \mathtt{B}$. From the premises of (T- INVK) we have that:

- $\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0$,
- $\Gamma \vdash \mathtt{e}_i : \mathtt{T}_i \ ^{(i\in 1..n)}$,
- $aType(\mathtt{C}_0)(\mathtt{m}) = \mathtt{A}_1 \cdots \mathtt{A}_n \to \mathtt{B}$, and
- $\mathtt{T}_i <: \mathtt{A}_i \ ^{(i\in 1..n)}$.

By induction we have that:

- $\Gamma \vdash \mathtt{e}_0 : \eta \mid \mathscr{E}_0$ and $\mathtt{CST} \models \mathscr{E}_0 \Rightarrow \mathbf{s}_0$ with $\mathbf{s}_0(\eta) = \mathtt{C}_0$, and
- $\Gamma \vdash \mathtt{e}_i : \tau_i \mid \mathscr{E}_i \ ^{(i\in 1..n)}$ and for all $i \in 1..n$: $\mathtt{CST} \models \mathscr{E}_i \Rightarrow \mathbf{s}_i$ with $\mathbf{s}_i(\tau_i) = \mathtt{T}_i$,

where, for all $i \neq j \in 0..n$, the set of the type variables in $\{\tau_i\} \cup \mathscr{E}_i$ and the set of the type variables in $\{\tau_j\} \cup \mathscr{E}_j$ (with $\tau_0 = \eta$) are pairwise disjoint. Consider

- $\alpha_1, ..., \alpha_n, \beta$ fresh, and
- $\mathscr{E} = \{\mathbf{meth}(\eta, \mathtt{m}, \alpha_1 \cdots \alpha_n \to \beta), \mathbf{subtype}(\tau_1\alpha_1), \ldots, \mathbf{subtype}(\tau_n\alpha_n)\}$.

From the rules in Fig. 8 we have that:

– $\texttt{CST} \models \mathbf{s}_n \circ \cdots \circ \mathbf{s}_0(\mathscr{E}) \Rightarrow \mathbf{s}'$ where $\mathbf{s}' = [\texttt{BA}_1 \cdots \texttt{A}_n/\beta\alpha_1 \cdots \alpha_n]$.

Then, by rule (CT- INVK), we get $\texttt{this} : \texttt{C}, \texttt{x}_1 : \texttt{A}_1, \ldots, \texttt{x}_p : \texttt{A}_p \vdash \texttt{e}_0.\texttt{m}(\texttt{e}_1, ..., \texttt{e}_n) : \beta \mid (\cup_{i\in\{0,...,n\}}\mathscr{E}_i) \cup \mathscr{E}$ and $\texttt{CST} \models \mathscr{E} \cup (\cup_{i\in\{0,...,n\}}\mathscr{E}_i) \Rightarrow \mathbf{s}' \circ \mathbf{s}_n \circ \cdots \circ \mathbf{s}_0$.
Cases (T- FIELD), (T- ASSIG), (T- UCAST), (T- DCAST) are similar to the previous case.

Lemma 7 *Let* CT *be a IFJ program,* $\texttt{C} \in dom(\texttt{CT})$, $\texttt{m} \in dom(\texttt{C})$, $\texttt{CST} = signature(\texttt{CT})$ *and* CST *satisfy the sanity conditions for class signature tables.*

(Soundness) Let $\texttt{this} : \texttt{C} \vdash \texttt{CT}(\texttt{C})(\texttt{m}) : \texttt{m}$ **with** $\mathscr{E}$ *and* $\texttt{CST} \models \mathscr{E}$. *Then* $\texttt{this} : \texttt{C} \vdash \texttt{CT}(\texttt{C})(\texttt{m})$ *OK.*
(Completeness) Let $\texttt{this} : \texttt{C} \vdash \texttt{CT}(\texttt{C})(\texttt{m})$ *OK. Then there exists* $\mathscr{E}$ *such that:* $\texttt{this} : \texttt{C} \vdash \texttt{CT}(\texttt{C})(\texttt{m}) : \texttt{m}$ **with** $\mathscr{E}$ *and* $\texttt{CST} \models \mathscr{E}$.

Proof Straightforward by Lemma 6.

Lemma 8 *Let* CT *be a IFJ program,* $\texttt{C} \in dom(\texttt{CT})$, $\texttt{CST} = signature(\texttt{CT})$ *and* CST *satisfy the sanity conditions for class signature tables.*

(Soundness) Let $\vdash \texttt{CT}(\texttt{C}) : \texttt{C}$ **with** $\mathscr{M}$ *and* $\texttt{CST} \models \text{FLAT}(\mathscr{M})$. *Then* $\vdash \texttt{CT}(\texttt{C})$ *OK.*
(Completeness) Let $\vdash \texttt{CT}(\texttt{C})$ *OK. Then there exists* $\mathscr{M}$ *such that:* $\vdash \texttt{CT}(\texttt{C}) : \texttt{C}$ **with** $\mathscr{M}$ *and* $\texttt{CST} \models \text{FLAT}(\mathscr{M})$.

Proof Straightforward by Lemma 7.

Restatement of Theorem 1 (Soundness and completeness of IFJ constraint-based typing) *Let* CT *be a* IFJ *program and* $\texttt{CST} = signature(\texttt{CT})$.

(Soundness) Let CST *satisfy the sanity conditions for class signature tables*, $\vdash \texttt{CT} : \mathscr{C}$ *and* $\texttt{CST} \models \text{FLAT}(\mathscr{C})$. *Then*

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

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

– $\vdash \texttt{CT} : \mathscr{C}$ and $\texttt{CST} \models \text{FLAT}(\mathscr{C})$, and
– if CT is cast-safe, then $\text{FLAT}(\mathscr{C})$ is cast-safe with respect to CST.

Proof *(Soundness)*

1. Straightforward by Lemma 8 (Soundness).
2. If $\text{FLAT}(\mathscr{C})$ is cast safe w.r.t. CST, then (T- DCAST) is not used.

(Completeness)

1. Straightforward by Lemma 8 (Completeness).
2. Observe that, if (T- DCAST) is not used, then $\text{FLAT}(\mathscr{C})$ is cast safe w.r.t. CST.

Appendix C: soundness and completeness of IFΔJ constraint-based typing

C.1 Proof of Proposition 2

Lemma 9 *For every delta module* δ *and for every class table* CT *such that* δ *is applicable to* CT, *and for every class* $\texttt{C} \in dom(\delta) \cap dom(\texttt{CT})$ *such that* $\vdash \delta(\texttt{C}) : cco$ *and* $\vdash \texttt{CT}(\texttt{C}) : cc$ *it holds that*

1. *for every method* $\mathtt{m} \in dom(\delta(\mathtt{C})) \cap dom(\mathtt{CT}(\mathtt{C}))$ *if*
   - $\mathtt{this} : \mathtt{C} \vdash \delta(\mathtt{C})(\mathtt{m}) : \{mco\}$, *and*
   - $\mathtt{this} : \mathtt{C} \vdash \mathtt{CT}(\mathtt{C})(\mathtt{m}) : \mathtt{m}$ **with** $\mathscr{E}$,

   then
   (a) $(\delta(\mathtt{C})(\mathtt{m}) = \mathtt{modifies}\ \cdots$ *and* $(mco = \mathtt{replaces}\ \cdots$ *or* $mco = \mathtt{wraps}\ \cdots))$ *or* $(\delta(\mathtt{C})(\mathtt{m}) = \mathtt{removes}\ \cdots$ *and* $mco = \mathtt{removes}\ \cdots)$,
   (b) $mco = \mathtt{replaces}\ \mathtt{m}$ **with** $\mathscr{E}'$ *implies that*
      i $\mathtt{this} : \mathtt{C} \vdash \text{APPLY}_\delta(\delta(\mathtt{C})\mathtt{CT}(\mathtt{C}))(\mathtt{m}) : \mathtt{m}$ **with** $\mathscr{E}'$,
      ii $\mathtt{m\$}\cdots \notin dom(\text{CONSAPPLY}_\delta(cco, cc))$,
   (c) $mco = \mathtt{wraps}\ \mathtt{m}$ **with** $\mathscr{E}'$ *implies that*
      i $\mathtt{this} : \mathtt{C} \vdash \text{APPLY}_\delta(\delta(\mathtt{C})\mathtt{CT}(\mathtt{C}))(\mathtt{m}) : \mathtt{m}$ **with** $\mathscr{E}'[\mathtt{m\$}\delta/\mathtt{original}]$,
      ii $\mathtt{this} : \mathtt{C} \vdash \text{APPLY}_\delta(\delta(\mathtt{C})\mathtt{CT}(\mathtt{C}))(\mathtt{m\$}\delta) : \mathtt{m\$}\delta$ **with** $\mathscr{E}$ *where* $cc(\mathtt{m}) = \mathtt{m}$ **with** $\mathscr{E}$,
   (d) $mco = \mathtt{removes}\ \mathtt{m}$ *implies that* $\mathtt{m} \notin dom(\text{CONSAPPLY}_\delta(cco, cc))$ *and* $\mathtt{m\$}\cdots \notin dom(\text{CONSAPPLY}_\delta(cco, cc))$;

2. *for every method* $\mathtt{m} \in dom(\delta(\mathtt{C})) - dom\mathtt{CT}(\mathtt{C})$ *if*
   - $\mathtt{this} : \mathtt{C} \vdash \delta(\mathtt{C})(\mathtt{m}) : \{mco\}$,

   then
   (a) $\delta(\mathtt{C})(\mathtt{m}) = \mathtt{adds}\ \cdots$ *and* $mco = \mathtt{adds}\ \cdots$,
   (b) $mco = \mathtt{adds}\ \mathtt{m}$ **with** $\mathscr{E}'$ *implies* $\mathtt{this} : \mathtt{C} \vdash : \text{APPLY}(\delta, \mathtt{CT})(\mathtt{C})(\mathtt{m})\mathtt{m}$ **with** $\mathscr{E}'$.

Proof Both points 1 and 2 follow straightforwardly by the definition of $\text{APPLY}(\delta\delta(\mathtt{C}))\mathtt{CT}(\mathtt{C})$ (given in Sect. 5.2) and the definition of $\text{CONSAPPLY}_\delta(cco, cc)$ (given in Sect. 7.2). By observing that rules (CT- S- ADDM), (CT- S- REPM) and (CT- S- WRAM) in Fig. 12 rely on rule (CT- METHOD) in Fig. 9.

Lemma 10 *For every delta module* δ *and for every class table* $\mathtt{CT}$ *such that* δ *is applicable to* $\mathtt{CT}$, $\vdash \mathtt{delta}\ \delta\ \cdots : \mathscr{D}$, *and* $\vdash \mathtt{CT} : \mathscr{C}$ *it holds that*

1. *for every class* $\mathtt{C} \in dom(\delta) \cap dom\mathtt{CT}$ *if*
   - $\vdash \delta(\mathtt{C}) : cco$, *and*
   - $\vdash \mathtt{CT}(\mathtt{C}) : \mathtt{C}$ **with** $\mathscr{M}$,

   then
   (a) $(\delta(\mathtt{C}) = \mathtt{modifies}\ \cdots$ *and* $cco = \mathtt{modifies}\ \cdots)$ *or* $(\delta(\mathtt{C}) = \mathtt{removes}\ \cdots$ *and* $cco = \mathtt{removes}\ \cdots)$,
   (b) $cco = \mathtt{modifies}\ \mathtt{C}$ **with** $\mathscr{O}$ *implies that* $\vdash \text{APPLY}_\delta(\delta(\mathtt{C})\mathtt{CT}(\mathtt{C})) : \text{CONSAPPLY}_\delta(\mathtt{modifies}\ \mathtt{C}$ **with** $\mathscr{O}, \mathtt{C}$ **with** $\mathscr{M})$,
   (c) $cco = \mathtt{removes}\ \mathtt{C}$ *implies that* $\mathtt{C} \notin dom(\text{CONSAPPLY}_\delta(\mathscr{D}, \mathscr{C}))$;

2. *for every class* $\mathtt{C} \in dom(\delta) - dom\mathtt{CT}$ *if*
   - $\vdash \delta(\mathtt{C}) : cco$

   then
   (a) $\delta(\mathtt{C}) = \mathtt{adds}\ \cdots$ *and* $cco = \mathtt{adds}\ \cdots$,
   (b) $cco = \mathtt{adds}\ cc$ *implies* $\vdash \text{APPLY}(\delta, \mathtt{CT})(\mathtt{C}) : cc$.

Proof Both points 1 and 2 follow straightforwardly by the definition of $\text{APPLY}_\delta(\delta(\mathtt{C})\mathtt{CT}(\mathtt{C}))$ (given in Sect. 5.2), the definition of $\text{CONSAPPLY}_\delta(cco, cc)$ (given in Sect. 7.2) and Lemma 9.

By observing that rule (CT- C- ADDC) in Fig. 12 relies on rule (CT- CLASS) in Fig. 9, and rule (CT- C- MODC) relies on rules (CT- S- ADDM), (CT- S- REPM) and (CT- S- WRAM) in Fig. 12.

Restatement of Proposition 2 *For every delta module* $\delta \in dom(\texttt{DMT})$ and for every class table CT such that δ is applicable to CT, if $\vdash \texttt{DMT}(\delta) : \mathscr{D}$ and $\vdash \texttt{CT} : \mathscr{C}$, then $\vdash \text{APPLY}(\delta, \texttt{CT}) : \text{CONSAPPLY}_\delta(\mathscr{D}, \mathscr{C})$.

Proof Straightforward by the definition of $\text{APPLY}(\delta, \texttt{CT})$ (given in Sect. 5.2), the definition of $\text{CONSAPPLY}_\delta(\mathscr{D}, \mathscr{C})$ (given in Sect. 7.2) and Lemma 10. By observing that rule (CT- DELTA) in Fig. 12 relies on rules (CT- C- ADDC) and (CT- C- MODC) in Fig. 12.

C.2: Proof of Theorem 2

Restatement of Theorem 2 (Soundness and completeness of IFΔJ constraint-based typing) *Let* L be a strongly unambiguous IFΔJ SPL and $\overline{\psi} \in \Phi$.

(Soundness) If $\texttt{CST}_{\overline{\psi}}$ is defined and satisfies the sanity conditions for class signature tables, $\vdash \texttt{delta}\ \delta\ \cdots : \mathscr{D}_\delta$ for all $\delta \in \Delta(\overline{\psi})$, 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 $\delta \in \Delta(\overline{\psi})$ 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 $\delta \in \Delta(\overline{\psi})$ such that δ is not $\vdash$-typable, then the body of the method-add/modify operation in δ that is ill typed is not included in the product $\texttt{CT}_{\overline{\psi}}$.

Proof (Soundness) Immediate by Corollary 2 and Theorem 1(Soundness).
(Completeness)

1. Immediate by Corollary 2 and Theorem 1(Completeness).
2. If the body of a method-add/modify operation in δ is ill typed, then it contains either an occurrence of a variable that is not a formal parameter of the method, or a stupid selection expression. Therefore, the inclusion of a method with a body containing either an occurrence of a variable that is not a formal parameter of the method or a stupid selection expression would contradict the assumption that $\vdash \texttt{CT}_{\overline{\psi}}$ OK.

References

1. Ancona, D., Damiani, F., Drossopoulou, S., Zucca, E.: Polymorphic bytecode: compositional compilation for java-like languages. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '05, pp. 26–37. ACM, New York (2005). doi:10.1145/1040305.1040308
2. Apel, S., Kästner, C., Grösslinger, A., Lengauer, C.: Type safety for feature-oriented product lines. Autom. Softw. Eng. **17**(3), 251–300 (2010). doi:10.1007/s10515-010-0066-8
3. Apel, S., Kästner, C., Lengauer, C.: Feature featherweight java: a calculus for feature-oriented programming and stepwise refinement. In: Proceedings of the 7th International Conference on Generative Programming and Component Engineering, GPCE '08, pp. 101–112. ACM, New York (2008). doi:10.1145/1449913.1449931

4. Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: An overview of caesarJ. Trans. AOSD I, LNCS **3880**, 135–173 (2006). doi:10.1007/11687061_5
5. Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) Software Product Lines (SPLC 2005), Lecture Notes in Computer Science, vol. 3714, pp. 7–20. Springer (2005). doi:10.1007/11554844_3
6. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE Trans. Softw. Eng. **30**, 355–371 (2004). doi:10.1109/TSE.2004.23
7. Bettini, L., Damiani, F., Schaefer, I.: Implementing software product lines using traits. In: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10, pp. 2096–2102. ACM, New York (2010). doi:10.1145/1774088.1774530
8. Bettini, L., Damiani, F., Schaefer, I., Strocco, F.: TraitRecordJ: a programming language with traits and records. Sci. Comput. Program. (2011). doi:10.1016/j.scico.2011.06.007
9. Bono, V., Damiani, F., Giachino, E.: Separating type, behavior, and State to achieve very fine-grained reuse. In: Electronic Proceedings of FTfJP (2007)
10. Bono, V., Damiani, F., Giachino, E.: On Traits and Types in a Java-like Setting. In: Ausiello, G., Karhumki, J., Mauri, G., Ong, L. (eds.) Fifth IFIP international conference on Theoretical Computer Science—Tcs 2008, IFIP International Federation for Information Processing, vol. 273, pp. 367–382. Springer (2008) doi:10.1007/978-0-387-09680-3_25
11. Bracha, G., Cook, W.: Mixin-based inheritance. In: Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA/ECOOP '90, pp. 303–311. ACM, New York (1990). doi:10.1145/97945.97982
12. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison Wesley Longman, Reading (2001)
13. Clifton, C., Leavens, G.T.: MiniMAO$_1$: investigating the semantics of proceed. Sci. Comput. Program. **63**(3), 321–374 (2006). doi:10.1016/j.scico.2006.02.009
14. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading (2000)
15. Damiani, F., Padovani, L., Schaefer, I.: A formal foundation for dynamic delta-oriented software product lines. In: Proceedings of the 11th International Conference on Generative Programming and Component Engineering, GPCE '12, pp. 1–10. ACM, New York (2012). doi:10.1145/2371401.2371403
16. Damiani, F., Schaefer, I.: Dynamic delta-oriented programming. In: Proceedings of the 15th International Software product Line Conference, vol. 2, SPLC '11, pp. 34:1–34:8. ACM, New York (2011). doi:10.1145/2019136.2019175
17. Damiani, F., Schaefer, I.: Family-based analysis of type safety for delta-oriented software product lines. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, Lecture Notes in Computer Science, vol. 7609, pp. 193–207. Springer (2012). doi:10.1007/978-3-642-34026-0_15
18. De Fraine, B., Südholt, M., Jonckers, V.: Strongaspectj: flexible and safe pointcut/advice bindings. In: Proceedings of the 7th International Conference on Aspect-Oriented Software Development, AOSD '08, pp. 60–71. ACM, New York (2008). doi:10.1145/1353482.1353491
19. Delaware, B., Cook, W., Batory, D.: A machine-checked model of safe composition. In: Proceedings of the 2009 Workshop on Foundations of Aspect-Oriented Languages, FOAL '09, pp. 31–35. ACM, New York (2009). doi:10.1145/1509837.1509846
20. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.: Traits: A mechanism for fine-grained reuse. ACM TOPLAS **28**(2), 331–388 (2006). doi:10.1145/1119479.1119483
21. Ernst, E.: gbeta - a language with virtual attributes, block structure, and propagating, dynamic inheritance. Ph.D. thesis, Department of Computer Science, University of Århus, Denmark (1999). http://www.daimi.au.dk/eernst/gbeta
22. Ernst, E.: Propagating class and method combination. In: Guerraoui, R. (ed.) ECOOP 1999—Object-Oriented Programming, Lecture Notes in Computer Science, vol. 1628, pp. 67–91. Springer (1999). doi:10.1007/3-540-48743-3_4
23. Ernst, E.: Higher-order hierarchies. In: Cardelli, L. (ed.) ECOOP 2003—Object-Oriented Programming, Lecture Notes in Computer Science, vol. 2743, pp. 303–328. Springer (2003). doi:10.1007/978-3-540-45070-2_14
24. Ernst, E.: The expression problem, Scandinavian style. In: MASPEGHI (2004). http://www.i3s.unice.fr/maspeghi2004/final-version/e_ernst.pdf
25. Fraine, B., Ernst, E., Sdholt, M.: Essential AOP: the a calculus. In: D'Hondt, T. (ed.) ECOOP 2010—Object-Oriented Programming, Lecture Notes in Computer Science, vol. 6183, pp. 101–125. Springer (2010). doi:10.1007/978-3-642-14107-2_6

26. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM TOPLAS **23**(3), 396–450 (2001). doi:10.1145/503502.503505
27. Johnsen, E., Kyas, M., Yu, I.: Dynamic classes: modular asynchronous evolution of distributed concurrent objects. In: Cavalcanti, A., Dams, D. (eds.) FM 2009: Formal Methods, Lecture Notes in Computer Science, vol. 5850, pp. 596–611. Springer (2009). doi:10.1007/978-3-642-05089-3_38
28. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report. Carnegie Mellon Software Engineering Institute (1990)
29. Kästner, C., Apel, S., Batory, D.: A case study implementing features using aspectJ. In: Software Product Line Conference (SPLC 2007), pp. 223–232. IEEE, Los Alamitos (2007). doi:10.1109/SPLINE.2007.12
30. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: Proceedings of the 30th International Conference on Software Engineering, ICSE '08, pp. 311–320. ACM, New York (2008). doi:10.1145/1368088.1368131
31. Kästner, C., Apel, S., ur Rahman, S.S., Rosenmüller, M., Batory, D., Saake, G.: On the impact of the optional feature problem: analysis and case studies. In: Proceedings of the 13th International Software Product Line Conference, SPLC '09, pp. 181–190. Carnegie Mellon University, Pittsburgh (2009). doi:10.1145/1753235.1753261
32. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectJ. In: ECOOP 2001—Object-Oriented Programming, Lecture Notes in Computer Science, vol. 2072, pp. 327–354. Springer (2001). doi:10.1007/3-540-45337-7_18
33. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP 1997—Object-Oriented Programming, Lecture Notes in Computer Science, vol. 1241, pp. 220–242. Springer (1997). doi:10.1007/BFb0053381
34. Krueger, C.: Eliminating the adoption barrier. IEEE Softw. **19**(4), 29–31 (2002). doi:10.1109/MS.2002.1020284
35. Kuhlemann, M., Batory, D., Kästner, C.: Safe composition of non-monotonic features. In: Proceedings of the Eighth International Conference on Generative Programming and Component Engineering, GPCE '09, pp. 177–186. ACM, New York (2009). doi:10.1145/1621607.1621634
36. Lopez-Herrejon, R., Batory, D., Cook, W.: Evaluating support for features in advanced modularization technologies. In: Black, A.P. (ed.) ECOOP 2005—Object-Oriented Programming, Lecture Notes in Computer Science, vol. 3586, pp. 169–194. Springer (2005). doi:10.1007/11531142_8
37. Madsen, O.L., Møller-Pedersen, B.: Virtual classes: a powerful mechanism in object-oriented programming. In: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, OOPSLA '89, pp. 397–406. ACM, New York (1989). doi:10.1145/74877.74919
38. Odersky, M.: The Scala Language Specification, version 2.4. Technical Report, Programming Methods Laboratory, EPFL (2007)
39. Ossher, H., Tarr, P.: Hyper/J: multi-dimensional separation of concerns for Java. In: Proceedings of the 22nd International Conference on Software Engineering, ICSE '00, pp. 734–737. ACM, New York (2000). doi:10.1145/337180.337618
40. Ostermann, K.: Dynamically composable collaborations with delegation layers. In: Magnusson, B. (ed.) ECOOP 2002—Object-Oriented Programming, Lecture Notes in Computer Science, vol. 2374, pp. 89–110. Springer (2002). doi:10.1007/3-540-47993-7_4
41. Rosenmüller, M., Siegmund, N., Saake, G., Apel, S.: Code generation to support static and dynamic composition of software product lines. In: Proceedings of the 7th International Conference on Generative Programming and Component Engineering, GPCE '08, pp. 3–12. ACM, New York (2008). doi:10.1145/1449913.1449917
42. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Bosch, J., Lee, J. (eds.) Software Product Lines: Going Beyond (SPLC 2010), Lecture Notes in Computer Science, vol. 6287, pp. 77–91. Springer (2010). doi:10.1007/978-3-642-15579-6_6
43. Schaefer, I., Bettini, L., Damiani, F.: Compositional type-checking for delta-oriented programming. In: Proceedings of the Tenth International Conference on Aspect-Oriented Software Development, AOSD '11, pp. 43–56. ACM, New York (2011). doi:10.1145/1960275.1960283
44. Schaefer, I., Damiani, F.: Pure delta-oriented programming. In: Proceedings of the 2nd International Workshop on Feature-Oriented Software Development, FOSD '10, pp. 49–56. ACM, New York (2010). doi:10.1145/1868688.1868696
45. Smaragdakis, Y., Batory, D.: Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. ACM Trans. Softw. Eng. Methodol. **11**(2), 215–255 (2002). doi:10.1145/505145.505148

46. Strniša, R., Sewell, P., Parkinson, M.: The java module system: core design and semantic definition. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, OOPSLA '07, pp. 499–514. ACM, New York (2007). doi:10.1145/1297027.1297064
47. Tarr, P., Ossher, H., Harrison, W., Sutton Jr., S.M.: N degrees of separation: multi-dimensional separation of concerns. In: Proceedings of the 21st International Conference on Software Engineering, ICSE '99, pp. 107–119. ACM, New York (1999). doi:10.1145/302405.302457
48. Thaker, S., Batory, D., Kitchin, D., Cook, W.: Safe composition of product lines. In: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, GPCE '07, pp. 95–104. ACM, New York (2007). doi:10.1145/1289971.1289989
49. Thüm, T., Apel, S., Kästner, C., Kuhlemann, M., Schaefer, I., Saake, G.: Analysis Strategies for Software Product Lines. Technical Report FIN-004-2012, School of Computer Science, University of Magdeburg, Germany (2012). http://www.cs.uni-magdeburg.de/inf_media/downloads/forschung/technical_reports_und_preprints/2012/04_2012.pdf
50. Torgersen, M.: The expression problem revisited. In: Odersky, M. (ed.) ECOOP 2004—Object-Oriented Programming, Lecture Notes in Computer Science, vol. 3086, pp. 123–146. Springer (2004). doi:10.1007/978-3-540-24851-4_6
51. Zenger, M., Odersky, M.: Independently extensible solutions to the expression problem. In: FOOL (2005)