Implementing Type-Safe Software Product Lines using Parametric Traits

Lorenzo Bettini\textsuperscript{a}, Ferruccio Damiani\textsuperscript{a,}\textsuperscript{*}, Ina Schaefer\textsuperscript{b}

\textsuperscript{a}Dipartimento di Informatica, Università di Torino
\textsuperscript{b}Institut for Software Engineering and Automotive Informatics, Technische Universität Braunschweig

Abstract

A software product line (SPL) is a set of related software systems with well-defined commonality and variability that are developed by reusing common artifacts. In this paper, we present a novel technique for implementing SPLs by exploiting mechanisms for fine-grained reuse which are orthogonal to class-based inheritance. In our approach the concepts of type, behavior, and state are separated into different and orthogonal linguistic concepts: interfaces, traits, and classes, respectively. We formalize our proposal by means of FEATHERWEIGHT PARAMETRIC TRAIT JAVA (FPTJ), a minimal core calculus where units of product functionality are modeled by parametric traits. Traits are a well-known construct for fine-grained reuse of behavior. Parametric traits are traits parameterized by interface names and class names. Parametric traits are applied to interface names and class names to generate traits that can be assembled in other (possibly parametric) traits or in classes that are used to build products. The composition of product functionality is realized by explicit operators of the calculus, allowing code manipulations for modeling product variability. The FPTJ type system ensures that the products in the SPL are type-safe by inspecting the parametric traits and classes shared by different products only once. Therefore, type-safety of an extension of a (type-safe) FPTJ SPL can be guaranteed by inspecting only the newly added parts.

Key words: Featherweight Java, Feature Model, Software Product Line, Trait, Type System

1. Introduction

A software product line (SPL) is a set of related software systems with well-defined commonality and variability [22, 55]. SPL engineering aims at developing these systems by managed reuse. Products of a SPL are commonly described in terms of features [35], where a feature is a unit of product functionality. Feature-based product variability has to be captured in the product line artifacts that are reused to realize the single products. On the implementation level, reuse mechanisms for product implementations have to be flexible enough to express the desired product variability. Additionally, they should provide static guarantees that the resulting products are type-safe. In order to be of effective use, the type-checking has to facilitate the analysis of newly added parts, if the product line evolves, without re-checking unmodified, already existing parts.

Today, most product implementations of SPLs are carried out within the object-oriented paradigm. Although class-based inheritance in object-oriented languages provides means for code reuse with static guarantees, the rigid structure of class-based inheritance puts limitations on the effective modeling of product variability and on the reuse of code (in particular, code reuse can be exploited only from within a class hierarchy) [49, 29]. Feature-oriented programming (FOP) [8] allows to flexibly implement product lines within the object-oriented paradigm by complementing class-based inheritance by class refinement. In FOP, a product implementation for a particular feature configuration is obtained by composing feature modules for the respective features. A feature module contains class definitions and

---

\textsuperscript{*}Corresponding Author.

Preprint submitted to Elsevier Friday 13th November, 2015
class refinements. A class refinement can modify an existing class by adding new fields/methods, by wrapping code around existing methods or by changing the superclass. Delta-oriented programming (DOP) [59, 61, 60] extends FOP by the possibility to remove code from an existing product. In DOP, a product implementation is obtained by applying modifications specified in delta modules to existing products. Both FOP [5, 28] and DOP [14] are equipped with type systems that allow establishing the type-safety of the implemented products.

In this paper, we explore another approach to structuring the implementation of SPLs in which flexible code reuse is combined with static guarantees. Instead of implementing products by specifying code modifications, products are realized by exploiting parametric traits. The term trait has been used by Ungar et al. [72], in the context of the dynamically-typed prototype-based language SELF, to refer to a parent object to which an object may delegate some of its behavior. Subsequently, Schärli et al. [62, 29] introduced traits in the context of the dynamically-typed class-based language SQUEAK/SMALLTALK, as means for fine-grained code reuse to overcome the limitations of class-based inheritance. A trait is a set of methods, completely independent from any class hierarchy. Parametric traits are traits parameterized by interface names and class names. In the original proposals of traits in SQUEAK/SMALLTALK [62, 29] (and in most of the subsequent formulation of traits within a JAVA-like nominal type system [65, 51, 58, 45, 44]) trait composition and class-based inheritance live together. However, class-based inheritance introduces an obstacle for flexibly implementing product lines since it limits the possibilities of reusing code. Therefore, in our approach, class-based inheritance is ruled out. Classes are assembled only by composition of code artifacts (traits and interfaces) that are suitable for reuse in different product implementations. By the trait parameterization mechanism and the trait modification operations, this programming language is particularly suitable to deal with unanticipated product line evolution.

We formalize our approach in Featherweight Parametric Trait Java (FPTJ), a minimal core calculus (in the spirit of FJ [34]) for interfaces, traits and classes. In FPTJ, the concepts of type, behavior, and state are separated into different and orthogonal linguistic concepts: interfaces, parametric traits and classes, respectively. FPTJ is an extension of the trait-based calculus presented in [18] with a trait parameterization mechanism, in order to better model product variability. The type system of FPTJ provides static guarantees on safe and consistent class assembly from traits and interfaces. The intent of this paper is twofold:

1. to introduce and formalize the trait parameterization mechanism, and
2. to illustrate how parametric traits support the development of software product lines, including unanticipated evolution of product lines.

In our approach, a product line consists of a code base and a product line declaration. The code base consists of a set of traits, interfaces and classes that form a well-typed FPTJ program. The product line declaration creates the connection to the product line variability specified in terms of product features [35]. Evolving the product line includes adapting the code base and the product line declaration.

The paper is organized as follows. In Section 2, we introduce parametric traits and illustrate them through a variant of the expression problem. In Section 3, we show how to use parametric traits to implement software product lines by considering the expression product line and its evolution. In Section 4, we present the FPTJ calculus and state its type soundness. In Section 5, we formalize FPTJ software product lines. Related work is discussed in Section 6. We conclude by outlining some directions for future work. The appendices contain proofs omitted from the main text.

A very preliminary version of the results presented in this paper has been presented in [13]. This paper contains a revised and improved version of the trait-based calculus, new examples and discussions, and the proofs of the main results. Concerning the calculus, we dropped the record construct [17, 15], we added trait parameterization for more appropriately modeling SPL variability, and we added the product line declaration construct.

2. Parametric Traits

Traits have been introduced in the dynamically-typed class-based language SQUEAK/SMALLTALK to play the role of units for fine-grained reuse of behavior; the common behavior (that is, the common methods) of a set of classes can be factored into a trait [62, 29]. Various formulations of traits in a JAVA-like setting can be found in the literature (see, e.g., [56, 50, 65, 51, 17, 58, 18, 45, 44, 15, 12, 10]). The programming language FORTRESS [1] (which does not contain class-based inheritance) incorporates a trait construct, while the “trait” construct of SCALA [52] is indeed a form of mixin (a mixin is a subclass parameterized over its superclass, see e.g., [20, 43, 32, 3]).
In this paper we build on the concept of trait as described in [18]. A trait can consist of provided methods, that implement behavior, of required methods, that parameterize the behavior itself, and of required fields, that can be directly accessed in the body of the provided methods. Traits are the building blocks to compose classes or other, more complex traits. A suite of trait composition operations allows the programmer to build classes and composite traits. A distinguished characteristic of traits is that the composite unit (class or trait) has complete control over conflicts that may arise during composition and must solve them explicitly (in case of name conflict a type error is immediately generated). Traits do not specify any state, therefore a class composed by using traits has to provide the required fields. The trait composition operations considered in this paper are as follows:

- A basic trait defines a set of methods and declares the required fields and the required methods.
- The symmetric sum operation, +, merges two traits to form a new trait. It requires that the summed traits must be disjoint (that is, they must not provide identically named methods).
- The operation exclude forms a new trait by removing a method from an existing trait.
- The operation aliasAs forms a new trait by giving a new name to an existing method.
- The operation renameTo creates a new trait by renaming all the occurrences of a required field name or of a required/provided method name from an existing trait.

Field and method requirements in traits are collected by the type system on a per-method basis. Namely, the constant-based type of each provided method \( m \) contains the information about which are the field requirements and method requirements that are used (i.e., selected on \( \text{this} \) in the body of \( m \)). So, since the constraint-based type of a trait is simply the set of the constraint-based types of its provided methods, the field requirements and method requirements that are not used are automatically dropped.

It is worth observing that the exclude and renameTo operations provide a great potential for unanticipated reuse. Namely, method exclusion supports the unanticipated removal of features and method/field reaming support the unanticipated renaming of features. The trait parameterization mechanism further increases the potential for unanticipated reuse by supporting the unanticipated change of the interface names and class names occurring in a trait.

This section illustrates parametric traits by considering the Expression Problem (EP) [73, 24, 71], an extensibility problem that has been proposed as a benchmark for data abstractions capable to support new data representations and operations. We consider a variant of the EP similar to the one presented by Zenger and Odersky [53]. We illustrate various incremental extensions of a base product, converging to an implementation of a datatype defined the grammar:

\[
\begin{align*}
\text{Exp} &::= \text{Lit} \mid \text{Add} \mid \text{Neg} \\
\text{Lit} &::= \langle \text{non-negative integers} \rangle \\
\text{Add} &::= \text{Exp} \cdot+ \cdot \text{Exp} \\
\text{Neg} &::= \text{"-" Exp}
\end{align*}
\]

with operations for: evaluating, which returns the value of the expression, and doubling, which returns a new expression which evaluates to a number which is the double the value of the argument expression. We use a JAVA-like notation and a more general syntax (including, e.g., the type \text{int}, the sequential composition operator, etc.) than the one of the FPTJ calculus presented in Section 4.

In a language with traits the methods of a class can be defined independently of the class itself, as illustrated by the code at the top of Listing 1. The class \text{LitEval} is composed from the interface \text{ILitEval} and the trait \text{TLitEval} as shown at the bottom of Listing 1. Parametric traits satisfy the so called flattening principle [29] (see also [51, 41]), that is, the semantics of a method introduced in a class by a trait is identical to the semantics of the same method...
defined directly within the class. For instance, the semantics of the class LitEval in Listing 1 is identical to the semantics of the JAVA class (note that our interfaces are literally JAVA interfaces):

class LitEval implements ILitEval {
    int value;
    ExpEval setLit(int v) { value = v; return this; }
    int eval() { return value; }
}

The body of a trait definition denotes the set of methods obtained by flattening it, that is, by replacing each occurrence of a trait name T by the set of methods denoted by the body of the definition of T and evaluating all trait composition operators (in a well-typed program the trait reuse graph must be acyclic, so the procedure is guaranteed to terminate). For example, the body of both the following trait definitions (where TLitEval refers to the trait definition in Listing 1)

trait T1 is TLitEval[exclude setLit, eval renameTo comp] + { boolean isLit() { return true; } }

trait T2 is {
    int value;
    int comp() { return value; }
    boolean isLit() { return true; }
}

denote the methods

int comp() { return value; }
boolean isLit() { return true; }

(with the field requirement int value). A parametric trait is a trait parameterized by interface names and class names. For instance, the definition

trait PTLitEval(interfaces Exp) is {
    int value;
    Exp setLit(int v) { value = v; return this; }
    int eval() { return value; }
}

defines a parametric trait PTLitEval with a formal interface parameter Exp. All the occurrences of the formal parameter name Exp in the body of the parametric definition are bound. Parametric traits can be applied to interface names and class traits names to generate traits. The body of a parametric trait definition denotes the same set of methods that would be denoted by the trait definition obtained by removing the parameters. The set of methods denoted by the application of a parametric trait PT is obtained by replacing the occurrences of the formal parameters with the corresponding actual parameters in the set of methods denoted by the body of PT (the constraint-based type system of the language performs the same substitution on the constraints to be checked when these methods will be used to define a class). For instance, the definition

trait TLitEval is PTLitEval(ExpEval)

is equivalent to the definition of the trait TLitEval given in Listing 1, since both the trait body

PTLitEval(ExpEval)

and the trait body

{ int value;
  Exp setLit(int v) { value = v; return this; }
  int eval() { return value; }
}

denote the methods

Exp setLit(int v) { value = v; return this; }
int eval() { return value; }

(with the field requirement int value). Also the trait body

TLitEval

denotes the same set of methods. This flattening semantics makes it possible to define parametric traits by exploiting in an unanticipated way (non-parametric) trait definitions. For instance, the definition of the parametric trait PTLitEval in the first line of Listing 4, where TLitEval refers to the trait definition in Listing 1, is equivalent to the definition of
Data Extensions. We now extend the data types by addition expressions and negation expressions. Listing 2 shows the code that is required to have an expression which is the addition of two expressions. The interface IAddEval extends the existing interface ExpEval by the respective object initialization method. The trait TAddEval contains the implementation of the initialization and eval methods for addition expressions. The class AddEval implements the addition expressions. Similarly, in Listing 3, negation expressions as implemented by the interface INegEval, the trait TNegEval and the class NegEval. The classes AddEval and NegEval can be deployed independently of each other. They can also live together to form a compound extension.

Operation Extensions. The trait parameterization mechanism supports to program unanticipated operation extensions. Listing 4 defines three parametric traits by exploiting the traits defined in Listings 1, 2, and 3, that were introduced without foreseeing that introducing corresponding definition parameterized by the interface name ExpEval in the first line of Listing 1.

Consider the problem to add an operation called twice to the expression data type which doubles the value of each literal in the expression, but maintains the overall structure of the expression. Listing 5 illustrates how the parametric traits introduced in Listing 4 provide a mean to achieve unanticipated operation extensions. First, we define two new interfaces where the interface ExpEvalTwice extends the existing interface ExpEval and the interface ILitEvalTwice extends the interface ExpEvalTwice with the initialization method of data type Lit. Trait TLitEvalTwice is built from the existing parametric trait PTLitEval by instantiating it with the new interface ExpEvalTwice to allow for adding the twice operation and composing it with a new anonymous trait providing the twice method. Class LitEvalTwice uses the newly defined interface ILitEvalTwice and the newly defined trait TLitEvalTwice. Note that a class name occurs in the implementation of the trait TLitEvalTwice. This can be made parametric such that also this trait can be reused as we will show in Section 3. The semantics of the class LitEvalTwice in Listing 5 is identical to the semantics of the JAVA class:

```
trait PTLitEval(interfaces ExpEval) is TLitEval
trait PTAddEval(interfaces ExpEval) is TAddEval
trait PTNegEval(interfaces ExpEval) is TNegEval
```
interface ExpEvalTwice extends ExpEval {
    ExpEvalTwice twice();
}

interface ILitEvalTwice extends ExpEvalTwice {
    ExpEvalTwice setLit(int v);
}

trait TLitEvalTwice is PTLitEval(ExpEvalTwice) + {
    int value;
    ExpEvalTwice twice() {
        return new LitEvalTwice().setLit(2 * value);
    }
}

class LitEvalTwice implements ILitEvalTwice by TLitEvalTwice {
    int value;
    int eval() {
        return value;
    }
    ExpEvalTwice twice() {
        return new LitTwice().setLit(2 * value);
    }
}

Listing 5: Artifacts for extending the product with features Lit,Eval to the product with features Lit,Eval,Twice

interface IAddEvalTwice extends ExpEvalTwice {
    ExpEvalTwice setAdd(ExpEvalTwice l, ExpEvalTwice r);
}

trait TAddEvalTwice is PTAddEval(ExpEvalTwice) + {
    ExpEvalTwice left;
    ExpEvalTwice right;
    ExpEvalTwice twice() {
        return new AddEvalTwice().setAdd(left.twice(), right.twice());
    }
}

interface INegEvalTwice extends ExpEvalTwice {
    ExpEvalTwice setNeg(ExpEvalTwice t);
}

trait TNegEvalTwice is PTNeg(ExpEvalTwice) + {
    ExpEvalTwice term;
    ExpEvalTwice twice() {
        return new NegEvalTwice().setNeg(term.twice());
    }
}

class AddEvalTwice implements IAddEvalTwice by TAddEvalTwice {
    ExpEvalTwice left; ExpEvalTwice right;
}

class NegEvalTwice implements INegEvalTwice by TNegEvalTwice {
    ExpEvalTwice term;
}

Listing 6: Artifacts for merging the product with features Lit,Add,Neg,Eval and the product with features Lit,Eval,Twice into the product with features Lit,Add,Neg,Eval,Twice

In the same way, we can extend the addition and negation expressions with a twice method. Listing 6 depicts the respective interfaces IAddEvalTwice and INegEvalTwice. For both, a new trait is built by instantiating the existing parametric trait with the new interface and adding a twice method via an anonymous trait. These traits are then used to build the classes AddEvalTwice and NegEvalTwice.

Encoding some standard class-based object oriented-programming mechanisms requires to write more verbose code. E.g., emulation of inheritance require an explicit method exclusion in order to write an overriding definition of a method, or a method renaming in order to provide access to the "super". However, the proposed trait composition operations provides better support for flexible creation of many variants of a software system.

The trait parameterization mechanism introduced in this paper aims to save the programmer from the burden to plan in advance the parametric trait definitions that might be useful for subsequent code extensions. However, when a new parametric trait is derived from an existing trait by turning class/interface names in the existing one to parameters, one name has to correspond to one parameter. So, it is not possible to replace different occurrences of the same name with different parameters. This represents a limitation of the mechanism.

3. Implementing Type-Safe Software Product Lines using Parametric Traits

SPL engineering is split into a family engineering and an application engineering phase [55]. During family engineering, the artifacts in the SPL artifact base are developed. In this setting, these are the reusable interfaces, traits and parametric traits. During application engineering, the artifacts are used for building the actual products by composition and instantiation of parameters. Figure 1 (left) shows the feature model [35] of the Expression Product Line (EPL) [46]. It separates the features of the EPL into two subgroups: data features and operations. The mandatory features are Lit and Eval which is denoted by a black dot. The optional features are the remaining features denoted by a white dot. The arrow from the feature Twice to the features Add and Neg with the stereotype requires denotes that the selection of the feature Twice also requires the selection of the other two features.
Product Line Specification. In our approach a product line consists of a code base (containing the reusable product line artifacts) and a product line declaration. The code base consists of a set of parametric traits, interfaces and classes. For instance, the code in Listings 1, 2, 3, 4, 5 and 6 represents a code base for the EPL. The product line declaration creates the connection to the product line variability specified in terms of product features. For instance, Listing 7 (left) shows a product line declaration for the EPL. The product line declaration:

- Lists the product features.
- Describes the set of valid feature configurations. In the examples, the valid feature configurations are represented by a propositional formula over the set of features. We refer to [7] for a discussion on other possible representations.
- Attaches to each class name a selection condition specifying for which feature configurations the class has to be included in the code of the corresponding product. In the examples, the selection condition is represented by a propositional constraint over the set of features, given by when clauses. Since only feature configurations that are valid according to the feature model are used for product generation, the selection conditions are understood as a conjunction with the formula describing the set of valid feature configurations.

A product is valid if it corresponds to a valid feature configuration. The product for a valid feature configuration can be automatically generated by taking the use-closure of the set of classes with a valid application condition, that is, by taking the set of classes with a valid application condition and adding all the used classes, parametric traits and interfaces.

Following [69], we say that a SPL is type safe if all of its (valid) products are well-typed programs (according to the type system of the language in which they are implemented). Therefore, since every use-closed subset of well-typed FPTJ program is well-typed, in order to ensure that a FPTJ product line is type safe it is enough to ensure that its code base is well typed.

Product Line Evolution. In practice, product lines are rarely planned ahead, but rather evolve to add more products following the principle of reactive product line engineering [39]. For product line evolution, we consider the extension of an existing product line by further products implementing a restricted set of features. These products can be derived
from existing products by removing features via restricting data or operations.1

Removal of operation features requiring operation restrictions is supported by the trait parameterization mechanism and the method exclusion operation. For instance, the product with features Lit, Eval, Twice can be implemented from the artifacts implementing the product with features Lit, Eval, Twice by dropping the class LitEvalTwice and adding the artifacts in Listing 8. In particular, the parametric trait PTLitEvalTwice is instantiated with the interface and class not containing the eval method and additionally the eval method itself is excluded.

The feature model for the Evolved Expression Product Line which also includes the product with features Lit, Eval, Twice, the product with features Lit, Add, Eval, Twice, the product with features Lit, Neg, Eval, Twice and the product with features Lit, Twice is given in the right part of Fig. 1. The code base of the extended product line is obtained by adding to code base of the original product line the artifacts in Listing 8, and the product line declaration is shown in Listing 7 (right).

Note that, when a class name occurs in a trait body, in order to typecheck the trait it is not needed to typecheck the body of the class, since it is enough to rely on the information provided by the interfaces implemented by the class. Therefore, the fact that the trait TLitTwice and the class LitTwice are mutually dependent (cf. Listing 8) does not represent a problem for the constraint-based type system that we will present in Section 4.4.

4. The FPTJ Calculus

In this section, we describe the syntax, semantics and typing of the FPTJ calculus (FEATHERWEIGHT PARAMETRIC TRAIT JAVA), a minimal core calculus (in the spirit of FJ [34]) for interfaces, parametric traits and classes that formalizes our proposal for using parametric traits to implement type-safe SPLs.

4.1. FPTJ Syntax

The syntax of FPTJ is given in Figure 2. We use the overbar sequence notation according to [34]. For instance, the pair “¯x” stands for “I1 x1, . . . , In xn”, and “¯ f,” stands for “I1 f1; . . . ; In fn;”. The empty sequence is denoted by “”. 2

In the FPTJ calculus (differently from most trait formulations, e.g., [65, 51, 58, 44, 52, 1]) trait names are not types and class names (although they are used as types by the type system) cannot be used as source level types (that is, they cannot be used as targets in typecast operations nor to declare the type of fields and methods). The only source level types are interface names. As pointed out in [18], using trait names as types limits the reuse potential of traits, because method exclusion and renaming operations would break the type system. Moreover, if class names are not used as source level types, interface declarations are independent from classes, and the dependencies of trait declarations on classes are restricted to object creation. Thus, by only using interfaces as source level types, the reuse potential of traits is increased. The trait parameterization mechanism further increases this reuse potential to appropriately capture product line variability.

In FPTJ there are no constructor declarations: in every class C we assume the implicit default constructor with no arguments that (like in JAVA) initializes all fields to null. Fields initialization (to values different from null)

---

1 Restriction of data features is straightforward, since it only requires to remove artifacts. For instance, the product with features Lit, Neg, Eval, Twice can be implemented straightforwardly by removing from the artifacts implementing the product with features Lit, Add, Neg, Eval, Twice the class AddEvalTwice, the interfaces IAddEval and IAddEvalTwice, and the traits TAddEval, PTAddEval and TAddEvalTwice.

2 Indeed, the observation that the role of type and the role of unit of reuse competing dates back at least to Snyder [66] (see also Cook et al. [23]).
is possible by using field assignment expressions. This approach is more suitable for programming SPLs than the explicit constructors of FJ, where all the fields must be initialized in a single constructor call whose parameters have to match the fields. Namely, using constructors a la FJ would make it difficult to deal with product transformations that add (or remove) fields (cf. the discussion in Sect. 3 of [28]).

Since traits do not introduce any state, a class has to provide the required fields of the traits it uses. Indeed, these field declarations could be avoided by adopting a design choice which states that, in a class assembled using traits, the required fields of the traits it uses are implicitly considered as provided. However, we have decided to require to declare the fields in the classes since we believe that these field declarations provide a better support for code documentation. Moreover, an IDE support can eliminate any burden to the programmer by a quickfix to automatically generate the declarations of all the fields required by the traits used by a class.

A class table $CT$ is a map from class names to class declarations. Similarly, an interface table $IT$ and a trait table $TT$ map interface names and trait names to interface and trait declarations, respectively. A FPTJ program $P$ is a triple $(IT, TT, CT)$. The free nominal types in a trait definition

\[
\text{trait } T(\text{classes } \bar{C}, \text{ interfaces } \bar{I}) \text{ is } TE
\]

are the interface names and the class names that occur in $TE$ and do not occur in the list of the formal parameters $(\text{classes } \bar{C}, \text{ interfaces } \bar{I})$. The free nominal types in a program $(IT, TT, CT)$ are the interface names and the class names that occur in $IT$ or in $CT$ or in the free nominal types in the trait definitions in $TT$. For the type system and the operational semantics, we assume fixed, global tables $IT$, $TT$, and $CT$. We also assume that these tables are well-formed, i.e., they contain an entry for each free nominal type and trait name occurring in the program, and that the interface subtyping and trait reuse graphs are acyclic.

Convention 4.1 (On Sequences of Named Elements). A sequence of named elements (e.g., interface declarations, method headers,...) is well-formed if it does not contain two (or more) elements with the same name. Sequences of named elements are in general assumed to be well-formed. The fact that a sequence of named elements $\bar{x}$ is well formed can be emphasized by writing "$\bar{x}$ is WF", e.g., in the premise of some typing rules. The sequence of names of the elements of $\bar{X}$ is denoted by names($\bar{X}$), the subsequence of the elements of $\bar{X}$ with the names $\bar{n}$ is denoted by choose($\bar{X}, \bar{n}$), and exclude($\bar{X}, \bar{n}$) denotes the sequence obtained from $\bar{X}$ by removing the elements with the names $\bar{n}$. According to [34], a set-based notation for operators over sequences of named elements is used. In the union and in the intersection of sequences, denoted by $\bar{X} \cup \bar{Y}$ and $\bar{X} \cap \bar{Y}$, respectively, it is assumed that if $n \in \text{names}(\bar{X})$ and $n \in \text{names}(\bar{Y})$ then choose($\bar{X}, n$) = choose($\bar{Y}, n$) ($n$ must be bound to an identical value in $\bar{X}$ and $\bar{Y}$). In the disjoint union of sequences, denoted by $\bar{X} \cdot \bar{Y}$, it is assumed that names($\bar{X}$) $\cap$ names($\bar{Y}$) = $\emptyset$.

4.2. On Flattening FPTJ to FFPTJ and Proving Type Soundness

Following a standard approach in the literature on traits [29] (see also [51, 41]), we specify the semantics of FPTJ by defining a "flattening" translation (that provides a canonical semantics for parametric traits by compiling them...
away) and by providing a semantics for the subset of the language without traits. To this aim, we introduce FFPTJ (FLAT FPTJ), the subset of FPTJ where there are no trait declarations and the syntax of trait expressions is simplified as follows:

\[
TE ::= \{ FD; \cdot; MD \}
\]

A FFPTJ class class C implements \( \hat{I} \) by \( \{ FD; \cdot; MD \} \) \( \{ FD; MD \} \) has the same meaning of the JAVA class class C implements \( \hat{I} \) \( \{ FD; MD \} \).

A FFPTJ program is a FPTJ program with an empty trait table, that is, a triple \((IT, \cdot, CT)\). The translation looks up named traits and evaluates all trait composition operators, removes the trait table, and replaces the class table with a class table containing only FFPTJ classes. Given a FPTJ program \( P \) we write \([P]\) to denote the corresponding FFPTJ program.

In Section 4.3 we present a standard type system for FFPTJ. This type system will be exploited to prove the soundness of the constraint-based type system for FPTJ, presented in Section 4.4. Namely, in Section 4.5, we will prove that if a FPTJ program \( P \) is typable by the constraint-based type system then the corresponding FFPTJ program \([P]\) is typable by the standard type system. Thus showing that the soundness of the standard type system for FFPTJ implies the soundness of the constraint-based type system for FPTJ.

The translation, specified by the function \( [\cdot]_T \) (in Figure 3), assumes that the FPTJ program to be translated is well typed according to the FPTJ constraint-based type system. Given a trait table \( T \), the translation maps a FPTJ class declaration \( CD \) to a FFPTJ class declaration \( [CD]_T \) and a trait expression \( TE \) the sequence \( [TE]_T \) of the methods it provides. We write \([CT]_T\) to denote the class table containing the translation of all the classes in \( CT \) with respect to the trait table \( T \). The translation of a FPTJ class assumes that all the fields of the class are used by some of the methods of the class (this is enforced by the type system). The translation of a basic trait expression ignores the required fields and the required method declarations. The clause for translating a parametric trait application \( T(\bar{D}) \) assumes that the number and kind of the actual parameters (the class names \( \bar{D} \) and the interface names \( \bar{J} \)) match the formal parameters (again, this is enforced by the type system). The remaining clauses for translating trait expressions are straightforward.

Summing two traits that provide a method with the same name, or aliasing a method as an already provided method, or renaming a method to an already provided method would cause the translation fail because of the disjoint union of sequences (cf. Convention 4.1), however, all these cases cannot occur when the program to be translated is well typed. Aliasing a method as a required method, or renaming a provided or required method to a required method, or renaming a required field to an already required field would never cause the translation to fail (moreover, if the program to be translated is well typed then types are guaranteed to match). The clause for translating field renaming is simpler than the clause for method renaming (which uses the auxiliary function \( mR \)); this is due to the fact that fields can be accessed only on this.

---

4To simplify the presentation, both the FFPTJ type system and the FPTJ constraint-based type system ensure that all the fields of a class are used by some of the methods of the class. Relaxing this check to accept FFPTJ classes of the form class C implements \( \hat{I} \) by \( \{ FD; \cdot; MD \} \) \( \{ FD; \} \) with \( FD \subseteq \{ FD; \} \) does not pose particular technical problems (it would only complicate a bit the case for classes in the definition of the flattening translation).
4.3. FFPTJ Typing

Nominal types are either class names or interface names. In order to type the null value (which is not considered in FJ [34]), the FFPTJ type system uses the special type \( \perp \), that is not a class or interface name and cannot occur in FFPTJ programs. An expression type (that is, a type that may be assigned to expressions) is either a nominal type or \( \perp \). The syntax of nominal types and expression types is given in Fig. 4.

The subtyping relation (Figure 5) between expression types extends the reflexive and transitive closure of the union of the immediate implements relation and extends relation (declared by the implements clauses in the class table \( CT \) and by the extends clauses in the interface table \( IT \), respectively) by ensuring that the type \( \perp \) is a subtype of any type. We write \( E_1 <: E_2 \) to mean that \( E_1 \) is a subtype of \( E_2 \).

The subtyping rules use the auxiliary functions fields, methods and \( mSig \), given in Fig. 6. The first two functions return the fields and the methods defined in a class \( C \), respectively. Note that the function methods is defined only for FFPTJ classes. The function \( mSig \) returns method signatures, ranged over by \( \sigma \) and \( \zeta \), i.e., method headers deprived of parameter names. For instance, the signature associated to the header \( \sigma_m(I_1 x_1, \ldots, I_n x_n) \) is \( \sigma_m(I_1, \ldots, I_n) \).

A type environment \( \Gamma \) is a mapping from variables (including this) to class names, written \( \text{this} : C \), \( \bar{x} : \bar{I} \). The empty environment is denoted by \( \bullet \).

The FFPTJ typing rules are given in Figure 7. We have a rule for typing the whole program (left implicit in FJ). The rule for interface definition \( (T\text{-INTERFACE}) \) exploits the auxiliary function \( mSig \) to ensure that, for every method name \( n \), all the method headers with name \( n \) listed in the body of the interface \( I \) and/or in any superinterface of \( I \) must have the same signature. The rule for class definition \( (T\text{-CLASS}) \) exploits the auxiliary function \( mSig \) to ensure that: (i) all the method definitions with name \( n \) listed in the body of the class and/or in any interface implemented by the class must have the same signature; and (ii) for any method declared in any interface implemented by the class there is a corresponding method definition in the class. The rules for method definition \( (T\text{-METHOD}) \), variable \( (T\text{-VAR}) \), object creation \( (T\text{-NEW}) \) and upcast \( (T\text{-UCAST}) \) are fairly standard. The rule for typing a cast that is not an upcast, \( (T\text{-NCAST}) \), models the fact that if the static type of \( e \) is an interface \( I_1 \), then \( e \) may evaluate to a reference to an object of a class that implements \( I \) even if \( I_1 \) is not a subtype of \( I \). The rule for field selection \( (T\text{-FIELD}) \) models the fact that all the fields are private to the object. The two rules for method invocation, \( (T\text{-INV1}) \) and \( (T\text{-INV2}) \), formalize the fact that the public methods of class are those listed in one of the interfaces implemented by the class, while the other methods are private to the object. We also have a rule for null and a rule for field assignment (not contained in FJ). Note that expressions like \( (1)e \) where the type of \( e \) is not a subtype of \( I \) (called stupid casts in [34]) or null.f and null.m(...)(that we call stupid selections) are ill-typed.\(^5\)

Note that, if a FFPTJ program is well-typed, then the program obtained from it by replacing every class declaration \( \text{class } C \text{ implements } I \) by \( \{ \text{FD; } \bullet; \text{ MD } \} \{ \text{FD; } \} \) with the JAVA class declaration \( \text{class } C \text{ implements } I \} \{ \text{FD; } \text{ MD } \} \) is literally a well-typed JAVA program.

The soundness of the FFPTJ type system w.r.t. a standard reduction semantics is illustrated in Appendix A.

4.4. FPTJ Constraint-based Typing

We have designed the FPTJ constraint-based type system driven by the following two requirements.

\(^5\)Indeed, stupid field selections (null.f) are not part of the syntax of FPTJ.
Fields lookup function (function fields)

$$\text{fields}(C) = \text{FD} \quad \text{if CT}(C) = \text{class } C \cdots \{ \text{FD} \}$$

Methods lookup function (function methods)

$$\text{methods}(C) = \text{MD} \quad \text{if CT}(C) = \text{class } C \cdots \text{by } \{ \cdots ; \text{MD} \} \{ \cdots \}$$

Method pure signatures lookup (function mSig)

$$\begin{align*}
\text{mSig}(\text{I m (I x)}) &= \text{I m (I)} \\
\text{mSig(MH}_{1, \ldots, \text{MH}_{n}}) &= \text{mSig(MH}_{1}) \cdots \text{mSig(MH}_{n}) \\
\text{mSig(I)} &= \text{mSig(I)} \cup \text{mSig(TE)} \\
\text{mSig(I)} &= \text{mSig(I)} \cup \text{mSig(TE)} \\
\text{mSig(MH} \{\text{return } e; \}} &= \text{mSig(MH)} \\
\text{mSig(MD}_{1} \ldots \text{MD}_{n}) &= \text{mSig(MD}_{1}) \cdots \text{mSig(MD}_{n}) \\
\text{mSig(C)} &= \text{mSig(I)} \quad \text{if CT(C) = class } C \text{ implements } I \text{ by } \cdots
\end{align*}$$

Figure 6: FPTJ: Functions fields, methods and mSig

1. The type assigned to any typable trait expression TE is the same type that it would be assigned to its flattened version \[\overline{\text{TE}}\] of TE.
2. Each parametric trait declaration TD (and therefore each trait expression TE) is typed by relying only on the constraint-based types inferred for the parametric trait declarations used by TD (resp. TE) and, in particular, independent of the declarations associated to the class names and interface names mentioned in TD (resp. TE).

The first requirement states that the type system must conform to the flattening semantics. The second requirement implies that typing a class declaration (or a parametric trait declaration) that uses a parametric trait T does not require to reinspect the declaration of T.\(^6\) Note that in the formulation of the second requirement we have explicitly stated that the typing of a trait declaration (and of a trait expression) must be independent from the declarations associated to the free types occurring in the trait declaration (resp. expression) since this condition is needed to deal with the trait parameterization feature.

4.4.1. Overview

In this section we illustrate the constraint-based typing judgements by pointing out that the constraint-based type system supports the two requirements presented at the beginning of Section 4.4.

The constraint-based type system infers a constraint-based type for each method definition in a trait. Each method MD \(\in\overline{\text{MD}}\) defined within a basic trait expression \(\langle \text{FD}; \text{MH}; \overline{\text{MD}} \rangle\) is type-checked by assuming for \(\text{this}\) the structural type \(\langle \text{FD} \cup \sigma \rangle\), where \(\text{FD}\) are the required fields and \(\sigma = \text{mSig(TE)} \cdot \text{mSig(TE)}\) are the signatures of the required and provided methods of the basic trait expression. Namely, the typing judgment for method definitions is \(\langle \text{FD} \cup \sigma \rangle \vdash_{\text{ct}} \overline{\text{MD}} : \mu\) to be read: “under the assumption that \text{this} has structural type \(\langle \text{FD} \cup \sigma \rangle\), the method declaration \(\overline{\text{MD}}\) has constraint-based type \(\mu\), where \(\mu = \text{I m (I)} \vdash \text{CT}\langle \text{FD} \cup \sigma \rangle\) is such that

1. \(\text{I m (I)}\) is the signature of the method;
2. the pair \(\langle \text{FD} \cup \sigma \rangle\) specifies that within the body of the method the fields \(\text{FD} \cup (\subseteq \text{FD})\) and the methods with signatures \(\overline{\sigma} (\subseteq \sigma)\) are selected on \(\text{this}\); and
3. \(\sigma\) is a set of constraints specifying subtyping checks, method signature looks up and typecast checks that must be satisfied in order to guarantee that the method is typable.

Therefore, field requirements (FD') and method requirements (\(\sigma'\)) are collected on a per-method basis and the field-/method requirements declared in a basic trait expression (\(\langle \text{FD}; \text{MH}; \overline{\text{MD}} \rangle\)) that are not used (i.e., not selected on \(\text{this}\)) in

---

\(^{6}\)As a matter of fact, some of the type systems for traits in Java-like nominal setting that has been proposed in the literature do not enjoy this property (see, e.g., the discussion in the seminal paper [65]).
Program typing  
\[ \vdash t \ \text{OK} \]
\[
\forall I \in \text{dom}(IT), \quad \vdash IT(I) \ \text{OK} \quad \forall C \in \text{dom}(CT), \quad \vdash CT(C) \ \text{OK} \\
\vdash (IT, \bullet, CT) \ \text{OK} \quad \tag{T\text{-}PROGRAM}
\]

Interface definition typing  
\[ \vdash t \ \text{ID} \ \text{OK} \]
\[
mSig(I) \ \text{wf} \\
\vdash t \ \text{interface} \ I \ \text{extends} \ J \ {\{M_H\}} \ \text{OK} \quad \tag{T\text{-}INTERFACE}
\]

Class definition typing  
\[ \vdash t \ \text{CD} \ \text{OK} \]
\[
\text{this:} \ C \vdash t \ \text{MD} \ \text{OK} \\
mSig(I) \subseteq mSig(MD) \\
\vdash t \ \text{class} \ C \ \text{implements} \ I \ \text{by} \ {\{F_D; \bullet; F_D; \}} \ {\{F_D; \}} \ \text{OK} \quad \tag{T\text{-}CLASS}
\]

Method definition typing  
\[ \text{this:} \ C \vdash t \ \text{MD} \ \text{OK} \]
\[
\text{this:} \ C, \ x: I \vdash t \ e: E \quad E : : I \\
\vdash t \ \text{this:} \ C \text{ im}(I)\{\text{return} \ e;\} \ \text{OK} \quad \tag{T\text{-}METHOD}
\]

Expression typing  
\[ \Gamma \vdash t \ e \]
\[
\Gamma \vdash x: \Gamma(x) \quad \tag{T\text{-}VAR}
\]
\[
\Gamma \vdash \text{this:} \ C \quad \text{If} \in \text{fields}(C) \\
\Gamma \vdash \text{this:} \ C \text{ im}(I)\{\text{return} \ e;\} \quad \tag{T\text{-}FIELD}
\]
\[
\Gamma \vdash \text{this:} \ C \quad \text{If} (I_1, \ldots, I_n) \in mSig(methods(C)) \quad \forall i \in 1..n. \quad \Gamma \vdash t_1: E_1 \quad E_i \ll : I_i \\
\Gamma \vdash t_1: \text{im}(I_1,\ldots,I_n): I \\
\Gamma \vdash e_0, t_0: \text{Im}(I_1,\ldots,I_n) \quad \forall i \in 1..n. \quad \Gamma \vdash t_1: E_1 \quad E_i \ll : I_i \\
\Gamma \vdash t_1: \text{m}(e_1,\ldots,e_n): I \\
\Gamma \vdash \text{new} \ C():C \quad \tag{T\text{-}NEW}
\]
\[
\Gamma \vdash t_1: E \quad E : : I \\
\Gamma \vdash (I) : : I \\
\Gamma \vdash (I)_e: I \\
\Gamma \vdash t_1: I \quad I_1 \ll : I \\
\Gamma \vdash (I)_e: I \\
\Gamma \vdash \text{null: } \bot \quad \tag{T\text{-}NULL}
\]
\[
\Gamma \vdash \text{this:} \ C \text{ im}(I)\{\text{return} \ e;\} \quad \Gamma \vdash t_1: E \quad E : : I \\
\Gamma \vdash \text{this:} \ C \text{ im}(I)\{\text{return} \ e;\} \quad \tag{T\text{-}ASSIG}
\]

Figure 7: FFPTJ: Typing rules programs, interfaces, classes, methods and expressions
the body of a provided method \( MD \in MD \) are automatically dropped from the constraint-based type \( \mu \) of \( MD \). Most of the constraints in the set \( C \) are needed to deal with trait parameterization. Namely, since in FPTJ any class or interface name occurring in a method definition \( MD \) may become a parameter in a parametric trait declaration that (re)uses (the basic trait expression containing) \( MD \), all the checks that involve a class or interface name have to be performed in the context of a class built by using the trait providing the method. In particular, if the trait parameterization feature is dropped from the language, then the set of constraint \( C \) can by replaced by the set of the interface names that (according to the use of \texttt{this} is the body of the method) must be implemented by the class of the \texttt{this} object.

The constraint-based type of a trait expression \( TE \) is the sequence of the constraint-based types of its provided methods. Namely, the typing judgement for trait expressions is \( \Gamma \vdash_{ct} TE : \hat{\mu} \) to be read: “the trait expression \( TE \) has constraint-based type \( \hat{\mu} \)”, where \( \hat{\mu} \) is the same type that it would be assigned to \( ||TE||_{TT} \).

The constraint-based type of a parametric trait declaration \( \text{trait } T \langle \text{classes } \check{C}, \text{interfaces } \check{I} \rangle \) is \( TE \) obtained by abstracting the names \( \check{C} \) and \( \check{I} \) from the type of its body \( TE \). Namely, the typing judgement for parametric trait declarations is \( \Gamma \vdash_{ct} \text{trait } T \langle \text{classes } \check{C}, \text{interfaces } \check{I} \rangle \) to be read: “the parametric trait declaration has constraint-based type \( \lambda \check{C}.\lambda \check{I}.\hat{\mu} \)”, where \( \hat{\mu} \) is the constraint-based type of the trait expression \( TE \). Then the rules for typing trait expressions can safely assign to the trait expression \( T(\check{D}) \) the (straightforwardly computed) type \( \hat{\mu}[\check{D}/\check{C}] \), since it is the same type that it would be assigned to \( ||T(\check{D})||_{TT} = ||TE||_{TT}[\check{D}/\check{C}] \).

Each class declaration \( C \text{ implements } \check{I} \) by \( TE \) is type-checked by verifying that all the constraints specified by constraint-based type of \( TE \) are satisfied. The typing judgment for class declarations is \( \Gamma \vdash_{ct} CD : \text{OK} \) to be read: “the class declaration \( CD \) is well typed”.

A constraint-based type environment \( \Delta \) is a type environment (cf. Section 4.3) where the type assumed for this is the type variable \texttt{thisClass} representing the name of the class of \texttt{this}. The use of the type variable \texttt{thisClass} is needed to make it possible to type a method body (which is an expressions) independently of the class declaration(s) that will incorporate the method. The typing judgment for expression is \( \Gamma, \varepsilon : \check{C} \vdash_{ct} \theta : \varepsilon \) to be read: “under the assumption that \texttt{this} has structural type \( \check{C} \) and the assumptions in \( \Delta \), the expression \( \varepsilon \) is well-typed with type \( \varepsilon \) modulo the constraints \( \langle \check{F}D \mid \check{\sigma} \rangle \) and \( \check{C} \).” The meaning of the constraints \( \langle \check{F}D \mid \check{\sigma} \rangle \) and \( \check{C} \) has been illustrated above (when introducing the typing judgement for method definitions). Note that in the premise of the judgement there are two distinct type assumption for this: the structural type assumption \( \langle \check{F}D \mid \check{\sigma} \rangle \) that is used to type method invocation on \texttt{this} and field selection and assignment; and the nominal type assumption \texttt{thisClass} that is used to type all the other uses of \texttt{this} in \( \varepsilon \).

The typing judgment for programs is \( \Gamma \vdash_{ct} (IT, TT, CT) : \text{OK} \) to be read: “the program \( (IT, TT, CT) \) is well typed”.

### 4.4.2. Constraints and Constraint Checking Rules

The constraints that may occur in the set \( C \) involve

- **open nominal types**, i.e., either nominal types or interface variables (ranged over by \( \alpha, \beta, \ldots \)), and
- **open expression types**, i.e., either expression types or interface variables or the distinguished type variable \texttt{thisClass}, representing the class of \texttt{this}.

A type variable is either an expression variable or \texttt{thisClass}. The constraints in the set \( C \) are checked w.r.t. a class declaration (cf. Section 4.4.1). Before checking the constraints the type variable \texttt{thisClass} is instantiated to the name of the class. During the checking of the constraints the interface variables are instantiated to interface names. The syntax of open nominal types and open expression types is illustrated in Figure 8 (top) and the syntax of the constraints that may occur in the set \( C \) is illustrated in Figure 8 (middle). An interface variable instantiation \( S \) is a mapping from interface variables to interface names. The instantiation that replaces the interface variables \( \check{I} \) by the interfaces names \( I \) (where \( I \) may contain duplicate names) is denoted by \( \check{I}/\check{I} \). The composition of two instantiations \( S_1 \) and \( S_2 \) with disjoint domain, which is a commutative operation, is denoted by \( S_1 \circ S_2 \).

The checking judgement for constraints is \( \Gamma \vdash_{ct} C : \check{S} \) to be read “the constraints in the set \( C \) are satisfied by the instantiation \( \check{S} \), where \( \check{C} \) does not contain occurrences of the type variable \texttt{thisClass} (before checking whether the constraints are w.r.t. a class declaration all the occurrences of \texttt{thisClass} must be replaced by the name of the class).” We write \( \Gamma \vdash_{ct} C : \text{OK} \) to mean that \( \Gamma \vdash_{ct} C : \check{S} \) holds for some instantiation \( \check{S} \). The associated rules are given in
Open nominal types and open types

| v ::= | N | α | open nominal types |
| ε ::= | E | α | thisClass | open expression types |

Constraints

\[ cns ::= \text{sub}(ε, I) | \text{cast}(I, ε) | \text{meth}(ν, m, αΣ) \]

\[ ε \text{ is a subtype of } I \]

\[ \text{type } ε \text{ can be casted to type } I \]

\[ \text{nominal type } ν \text{ has method } m \text{ with signature } α m(Σ) \]

Rules for checking constraints satisfaction

\[ \vdash_{ce} ε : S \]

(CC-EMPTY) \[ \vdash_{ce} ε : S \]

(CC-SUB) \[ E <: I \vdash_{ce} ε : S \]

(CC-CAST) \[ (E <: I \text{ or } E \text{ is an interface name}) \vdash_{ce} ε : S \]

(CC-METH) \[ I m(\overline{I}) ∈ mSig(\overline{N}) \vdash_{ce} ε[I/\overline{I}] : S \]

Figure 8: Open nominal types and open expression types syntax (top), constraints syntax (middle) and constraint checking rules (bottom)

Figure 8 (bottom); the operator \( \uplus \) denotes the disjoint rules of set of constraints. The rules are almost self explanatory, according to the informal meaning given in the middle of Figure 8. In particular, rule (CC-EMPTY) states that the empty set of constraints is satisfied by the empty instantiation, rule (CC-SUB) and (CC-CAST) rely on the subtyping relation \(<:\) (introduced in Section 4.3) and rule (CC-METH) relies on the signature lookup function \( mSig \) (given in Figure 6 of Section 4.3). We say that a constraint is \( ground \) to mean that it contains no type variables. The checking of a constraint of the form \( \text{sub}(\cdot, \cdot) \) or \( \text{cast}(\cdot, \cdot) \) can be performed only when the constraint is ground. The checking of a constraint of the form \( \text{meth}(\cdot, \cdot, \cdot) \) can be performed only when the first argument is a nominal type (that is, either a class name or an interface name) the last argument contains only interface variables; the checking causes the instantiation of all the interface variables occurring in the third argument.

4.4.3. Constraint-based Typing Rules for Programs, Parametric Traits, Classes, Basic Trait Expressions, and Methods

The constraint-based typing rules for programs, parametric trait and class declarations, for basic trait expressions, and for methods definitions are given in Fig. 9. The constraint-based typing rule for interface declaration is the same as in the type system for FFPTJ (rule (T-INTERFACE) in Fig. 7) and the constraint-based typing rule for programs (CT-PROGRAM) explicitly relies on the FFPTJ typing rule for interfaces.

The constraint-based typing rule for parametric trait declarations (CT-TRAIT) assigns to

\[ \text{trait } T(\text{classes } \overline{C}, \text{ interfaces } \overline{I}) \text{ is } TE \]

the type \( λ \overline{C}.λ \overline{I}.μ \) where \( μ \) is the type of the trait expression \( TE \). Since the trait reuse graph is acyclic (cf. end of Section 4.1) no use of \( T \) may be encountered when typing \( TE \). The constraint-based typing rule for class declarations (CT-CLASS) considers

\[ \text{class } C \text{ implements } \overline{I} \text{ by } TE \{ FD \} \]

and checks that the constraints in the types of the methods provided by \( TE \) are satisfied. Namely, that

1. subtyping, method signature lookup and typecast checks required by the methods provided by \( TE \) (collected in the sets \( ε_i \)) are satisfiable if the distinguished type variable \( \text{thisClass} \) is instantiated to \( C \),
2. the class \( C \) provides the fields required by \( TE \), and
3. \( TE \) provides all the methods it requires and all the methods in the interfaces implemented by \( C \).

The constraint-based typing rule for basic trait expressions (CT-TEBASIC) assigns to

\[ \{ FD; MH; MD_1...MD_p \} \]
the sequence of the types inferred for the method definitions $M_D i (1 \leq i \leq p)$ by assuming the structural type $\{ \text{FD} : mSig(\mathbb{P}) \} \cdot mSig(\mathbb{M}_D ... \mathbb{M}_p)$ for $\text{this}$. The last two premises check that each required field declaration in $\text{FD}$ and each required method declaration in $\mathbb{P}$ is used by some of the provided methods $\mathbb{M}_D ... \mathbb{M}_p$. Rule (CT-TEBASIC) concerns the typing for trait expressions, so it should have been presented together with the other rules for trait expressions (in Figure 10). We have decided present it separately since it is the only typing rule for trait expressions that is needed to deal with the flat subset of the language (FFPTJ).

The typing rule for method definitions (CT-METHOD) assigns to

$I \text{m} (I \bar{x}) \{ \text{return } e; \}$

the constraints inferred for the body of the method $e$ augmented with a constraint expressing that the type of $e$ must be a subtype of $I$ (the declared return type of the method).

4.4.4. Typing Rules for Non-Basic Trait Expressions

The typing rules for non-basic trait expressions are given in Fig. 10. The rule for parametric trait application (CT-TENAME) looks up the typing $\lambda \mathcal{C} \lambda \mu \mathcal{P}$ of the declaration of the parametric trait $\mathcal{T}$ and assigns to $T(\mathcal{D} \mathcal{J})$ the typing $\mu(\mathcal{D} \setminus \mathcal{C})$.

The rule for method exclusion (CT-TEEXCLUDE) simply removes the type of the excluded method. Since the typing rules collect field and method requirements on a per-method method basis (cf. explanation at the beginning of Section 4.4.1), the type $\mu$ of each method $m$ provided by the trait expression $TE$ contains the information about which are

\[ \frac{\forall i \in dom(CT), \quad \vdash_{ct} CT(TE)} {\vdash_{ct} CT(c)} \quad \text{(CT-PROGRAM)} \]

\[ \frac {\vdash_{ct} \mu \mathcal{C} \lambda \mathcal{P}} {\vdash_{ct} \mu \mathcal{C} \lambda \mathcal{P}} \quad \text{(CT-TRAITS)} \]

\[ \frac {\forall i \in 1 ... p, \quad \mu_i = \zeta_i \cdot (\text{FD}(i) : \sigma(i)) + \mathcal{C}_i \quad \vdash_{ct} \mathcal{C}_i [\text{thisClass}] \quad \text{OK}} {\text{FD} = \cup_{i \in 1 ... p} \text{FD}(i) \quad \sigma = \text{exclude}((\cup_{i \in 1 ... p} \sigma(i)) \cdot \text{names}(\zeta_1 ... \zeta_p)) \quad \vdash_{ct} \mathcal{C} \mathcal{C}_i \cdot \mathcal{P}} \quad \text{(CT-CLASS)} \]

\[ \frac {\forall i \in 1 ... p, \quad \mu_i = \zeta_i \cdot (\text{FD}(i) : \sigma(i)) + \mathcal{C}_i \quad \text{FD} = \cup_{i \in 1 ... p} \text{FD}(i) \quad \sigma = \text{exclude}((\cup_{i \in 1 ... p} \sigma(i)) \cdot \text{names}(\zeta_1 ... \zeta_p)) \quad \vdash_{ct} \mathcal{C} \mathcal{C}_i \cdot \mathcal{P}} {\vdash_{ct} \mathcal{C} \mathcal{C}_i \cdot \mathcal{P}} \quad \text{(CT-TEBASIC)} \]

Figure 9: FPTJ: Constraint-based typing rules for programs, classes, parametric traits, basic trait expressions and methods
(Non-basic) trait expression typing

\[
\begin{align*}
\vdash \text{ct} \, \text{TE} : \tilde{\mu} \\
\vdash \text{ct} \, \text{trait} \, T : \ldots : \lambda \, \tilde{\lambda} \, \tilde{\lambda} \, \tilde{\mu} \\
\vdash \text{ct} \, \text{T}(\tilde{\theta}) : \mu[\tilde{\mathcal{D}} \cup \tilde{\mathcal{C}}][\tilde{\mathcal{I}}] \quad \text{(CT-TNAME)}
\end{align*}
\]

\[
\begin{align*}
\forall i \in 1..p + q, \quad \mu_i = \zeta_i & : (\mathcal{F}_D^{(i)} \cup \mathcal{S}^{(i)}) \cup \mathcal{C}_i \cup \bigcup_{i \in 1..p+q} \mathcal{F}_D^{(i)} \quad \text{\textbf{wf}} \\
\vdash \text{ct} \, \text{TE}_1 : \mu_1 \ldots \mu_p & \quad \vdash \text{ct} \, \text{TE}_2 : \mu_{p+1} \ldots \mu_{p+q} \quad \vdash \text{ct} \, \text{TE}_1 \cup \text{TE}_2 : \mu_1 \ldots \mu_{p+q} \quad \text{\textbf{wf}} \\
\forall i & : \alpha_{\mathbf{ renaming}} \vdash \text{ct} \, \text{TE} : \mu \cdot \mu' & \quad \text{\textbf{names}}(\mu) = n \quad \text{(CT-TECHANGE)}
\end{align*}
\]

\[
\begin{align*}
\vdash \text{ct} \, \text{TE} : \mu_1 \ldots \mu_n & \quad n \geq p \geq 1 \quad \forall i \in 1..n, \quad \mu_i = \zeta_i : (\mathcal{F}_D^{(i)} \cup \mathcal{S}^{(i)}) \cup \mathcal{C}_i \\
\mu & = \zeta : (\mathcal{F}_D^{(p)} \cup \mathcal{S}^{(p)}) \cup \mathcal{C}_p \quad \text{\textbf{wf}} \\
\exists m \in \text{names}(\zeta) & \quad \mathcal{F}_{\text{ aliasAs}} \, m : \mu_1 \ldots \mu_p \quad \text{(CT-TEALIAS)}
\end{align*}
\]

\[
\begin{align*}
\tilde{\zeta} = \zeta_1 \ldots \zeta_n & \quad \forall i \in 1..n, \quad \mu_i = \zeta_i : (\mathcal{F}_D^{(i)} \cup \mathcal{S}^{(i)}) \cup \mathcal{C}_i \\
\tilde{\mu} & = \zeta : (\mathcal{F}_D^{(p)} \cup \mathcal{S}^{(p)}) \cup \mathcal{C}_p \quad \text{\textbf{wf}} \\
\vdash \text{ct} \, \text{TE}[\text{renameTo} m'] : \mu_1 \ldots \mu_p & \quad \text{(CT-TENAMEM)}
\end{align*}
\]

\[
\begin{align*}
\exists \tilde{f} \in \text{names}(\mathcal{F}_D^{(1)} \cup \ldots \cup \mathcal{F}_D^{(p)}) & \quad \forall i \in 1..n, \quad \mu_i = \zeta_i : (\mathcal{F}_D^{(i)} \cup \mathcal{S}^{(i)}) \cup \mathcal{C}_i \\
\tilde{f} \in \text{names}(\mathcal{F}_D^{(1)} \cup \ldots \cup \mathcal{F}_D^{(p)}) & \quad \mathcal{F}_D^{(i)} = (\text{exclude}(\mathcal{F}_D^{(i)}, \mathcal{W}[\mathcal{M}]) \cup (\text{choose}(\mathcal{F}_D^{(i)}, \mathcal{W}[\mathcal{M}]) \quad \text{\textbf{wf}} \\
\vdash \text{ct} \, \text{TE}[\text{renameTo} \tilde{f}'] : \mu_1 \ldots \mu_p & \quad \text{(CT-TERENAMEF)}
\end{align*}
\]

Figure 10: FPTJ: Constraint-based typing rules for non-basic trait expressions

the fields and methods that are selected on this in the body of m. Therefore, when a method m is excluded, there is no need to update the information about required methods in the types of the remaining methods.

The rule for symmetric sum of traits (\textbf{CT-TESUM}) checks (by the two \textbf{wf} statements in its premises) that there are no conflicts among the fields required by the summed traits and among provided methods (\zeta_1 \ldots \zeta_{p+q}) and required methods (\bigcup_{i \in 1..p+q} \mathcal{S}^{(i)}).\textsuperscript{8} Then it assigns to the composed trait the type resulting from the concatenation of the types of the summed traits.

The rule for method aliasing (\textbf{CT-TEALIAS}) besides ensuring that the method to be aliased exists, it also checks that the new name does not create conflicts. The type of the alias method is added to the final type.

The rule for method renaming (\textbf{CT-TENAMEM}) ensures that the method to be renamed is either provided or required or both and ensures that the new name does not create conflicts (that is, m' was not already provided and, if it was already required, then it has the same type of n). Then the method name substitution is performed on the signatures of both the required and the provided methods.

The rule for field renaming (\textbf{CT-TERENAMEF}) ensures that the field to be renamed is required and that the new field name does not create conflicts (that is, if \tilde{f}' was already required then it has the same type of f). Then the field name substitution is performed on the field requirements.

4.4.5. Typing Rules for Expressions

The typing rules for expressions are given in Fig. 11. The rules are syntax directed, with one rule for each term. The rule for variables (\textbf{CT-VAR}) is fairly standard; it looks up the type of x in \Delta and no constraints on this have to be

\textsuperscript{8}According to Convention 4.1, mentioning the sequence of method signatures \zeta_1 \ldots \zeta_{p+q} implies that there are no conflicts between the signatures of the methods provided by the summed traits.
Expression typing

\[
\begin{align*}
\text{Expression typing} & \quad \left\langle \text{FD} \mid \sigma \right\rangle; \Delta \vdash e : e' \quad \left( \text{CT-ASSIG} \right) \\
\left\langle \text{FD} \mid \sigma \right\rangle; \Delta \vdash x : \Delta(x) \quad \left( \text{CT-VAR} \right) \\
\end{align*}
\]

\[
\begin{align*}
\text{choose}(\text{FD}, f) & = \text{If} \\
\left\langle \text{FD} \mid \sigma \right\rangle; \Delta \vdash \text{this} \cdot f : I \quad \left( \text{CT-FIELD} \right) \\
\end{align*}
\]

\[
\begin{align*}
\forall i \in 1..n, \quad \left\langle \text{FD} \mid \sigma \right\rangle; \Delta \vdash e_i : \left( \text{FD}^{(i)} \mid \sigma^{(i)} \right) \quad \left( \text{CT-INV} \right) \\
\end{align*}
\]

\[
\begin{align*}
\text{choose}(\sigma, m) & = \text{Im}(I_1 \ldots I_n) \\
\end{align*}
\]

\[
\begin{align*}
\left\langle \text{FD} \mid \sigma \right\rangle; \Delta \vdash \text{this} \cdot m(e_1 \ldots e_n) : I \quad \left( \text{CT-ASSIG} \right) \\
\end{align*}
\]

\[
\begin{align*}
\alpha_1 \ldots \alpha_n \quad \text{fresh} & \quad (C : C) : \left( \text{CT-NEW} \right) \\
\end{align*}
\]

\[
\begin{align*}
\left\langle \text{FD} \mid \sigma \right\rangle; \Delta \vdash \text{null} : \bot \quad \left( \text{CT-NULL} \right) \\
\end{align*}
\]

\[
\begin{align*}
\text{choose}(\text{FD}, f) & = \text{If} \\
\left\langle \text{FD} \mid \sigma \right\rangle; \Delta \vdash e : e' \quad \left( \text{CT-CAST} \right) \\
\end{align*}
\]

\[
\begin{align*}
\left\langle \text{FD} \mid \sigma \right\rangle; \Delta \vdash e : \left( \text{FD}^{'} \mid \sigma^{'} \right) \quad \left( \text{CT-ASSIG} \right) \\
\end{align*}
\]

Figure 11: FPTJ: Constraint-based typing rules for expressions

collected (even when \( x \) is \text{this}).

The rule for field selection (\text{CT-FIELD}) extracts from the structural type \( \left\langle \text{FD} \mid \sigma \right\rangle \) assumed for \text{this} the type \( I \) of the selected \( f \) and collects the constraint that this must have a field \( f \) of type \( I \). The constraints collected by means of rule (\text{CT-FIELD}) are a subset of the assumptions \( \text{FD} \): they describe the fields that are selected on \text{this} by the checked expression. Collecting this precise information (instead of the whole \( \text{FD} \)) makes it possible to check trait expressions \( \text{TE} \) by checking the fields that are effectively required by the provided methods. Such fields, due to the presence of the method exclusion operation, can be a subset of the fields requirements declared in the basic trait expressions used by \( \text{TE} \).

In the rule for method invocation when the receiver is the distinguished variable \text{this} (\text{CT-INV}) extracts from the structural type \( \left\langle \text{FD} \mid \sigma \right\rangle \) assumed for \text{this} the type signature of the invoked method \( m \) and collects a constraint expressing that this must have a method \( m \) with that signature. The actual parameters \( e_1 \ldots e_n \) are checked and the inferred constraints are collected in the conclusion of the rule together with the constraints expressing that the type of each actual parameters must be a subtype of the type of the corresponding formal parameter. The rule for method invocation when the receiver \( e \) is not the distinguished variable \text{this} (\text{CT-INV}) is similar modulo the fact that (due the presence of trait parameterization in FPTJ any class or interface name occurring in a method definition may become a parameter in a subsequent use of the basic trait expression containing the method declaration) there is no way to retrieve the signature of the invoked method \( m \). The issue is solved by introducing fresh interface variables and by collecting constraints expressing that the type of the receiver must have a method of name \( m \) with the right number of formal parameters and that the type of each actual parameters must a subtype of the type of the corresponding formal parameter. Rules (\text{CT-NEW}), (\text{CT-CAST}), (\text{CT-NULL}) and (\text{CT-ASSIG}) do not present particular difficulties.

4.5. FPTJ Type Soundness

The soundness of the FPTJ constraint-based typing is proved in two steps. First, we show that the flattening translation preserves types. Second, we prove that for FFPTJ programs the constraint-based type system (illustrated
Theorem 4.2. (Flattening Preserves the Type of Programs)  
If \( \vdash_{ct} (IT, TT, CT) \) OK, then \( \vdash_{ct} (IT, \bullet, [CT]_{TT}) \) OK.

\[ \text{PROOF. See Appendix B.} \]

Theorem 4.3. (Equivalence of \( \vdash_{ct} \)-typability and \( \vdash_{t} \)-typability on FFPTJ programs) For every FFPTJ program \( P = (IT, \bullet, CT) \) such that

\[
\begin{align*}
IT(IMain) &= \text{interface IMain extends } \bullet \{ \text{main(); }, \text{ and } \\
CT(CMain) &= \text{class CMain implements IMain by } \{ \text{main() } \{ \text{return e;} \} \{ \bullet \}
\end{align*}
\]

it holds that

1. \( \vdash_{ct} P \) OK if and only if \( \vdash_{t} P \) OK
2. \( (\bullet ; \bullet ); \bullet \vdash_{ct} e : E \iff (\bullet ; \bullet ); \vdash_{t} e : E \) where both \( \vdash_{ct} E : S \) and \( \varepsilon S = E \) hold if and only if \( \bullet \vdash_{t} e : E \).

\[ \text{PROOF. See Appendix C.} \]

Consider the constraint-based typing rules for expressions in Figure 11. The only rule that creates interface variables is \((CT-I\text{-INV}K2)\) and the interface variables created by each application of the rule are listed in the third argument of the collected constraint \( \text{meth}(\nu, m, \alpha_1, \cdots, \alpha_n) \). Therefore, the checking rules for constraints (given in Section 4.4.2) can be applied by considering the constraints in the order in which they are created. In fact:

1. the check of any constraint \( \text{meth}(\cdots, \cdots, \cdots) \) can be performed only when its first arguments is a nominal type its third arguments contains interface variables only;
2. performing the check causes the instantiation of all the interface variables occurring in the constraint; and
3. trying to check the constraints in a different order cannot cause a different instantiation of any interface variable.

The constraints collected by the FPTJ type system can be exploited to determine the exact location in a used parametric trait causing a type error in class declaration (or in another parametric trait declaration). The idea (not formalized in current presentation of the type system) is to record for each generated constraint the location of the associated code in the parametric trait declarations.

5. Formalizing Type-safe Software Product Lines in FPTJ

In this section, we formalize FPTJ software product lines. We use the metavariables \( \phi \) and \( \psi \) to range over feature names. We write \( \bar{\phi} \) as short for the set \( \{ \phi \} \), i.e., the feature configuration containing the features \( \phi \). An FPTJ SPL is a 4-tuple \( L = (P, \bar{\phi}, \Phi, W) \) consisting of:

1. a FPTJ program \( P = (IT, TT, CT) \),
2. the features \( \bar{\phi} \) of the SPL,
3. the set of the valid feature configurations \( \Phi \subseteq \mathcal{P}(\bar{\phi}) \),\(^9\)
4. a mapping \( W : \Phi \rightarrow \mathcal{P}(\text{dom}(CT)) \) determining, for each feature configuration, the classes that are selected to build the product (this mapping represent the information provided by the when clauses used in the examples presented in Section 3).

\(^9\)We abstract from the concrete representation of the feature model.
The 3-tuple \((\varphi, \Phi, W)\) represents the product line declaration, while the FPTJ program \(P\) is the code base. The product associated to a feature configuration \(\psi \in \Phi\) is the use-closure of the classes in \(W(\psi)\).\(^{10}\) Therefore, in order to ensure that a FPTJ product line is type safe (that is, all the products are well typed) it is enough to check that its code base is a well-typed FPTJ program.

The FPTJ trait composition and parameterization mechanisms is particular suitable for supporting a lightweight evolution of a type-safe SPL consisting of the following three steps:

1. adding new artifacts to the product line code base by ensuring that type safety is preserved (this can be done by inspecting only the newly added parts),
2. changing the product line declaration, and
3. (possibly) removing useless artifacts.

This lightweight evolution does not require to type check again already existing parts of the product line and is particularly well suited for reactive SPL development. In contrast, an SPL evolution that requires to modify existing artifacts also requires to type-check again the already existing unchanged artifacts that use the modified artifacts. Two examples of this kind of evolution are refactoring (that is, restructuring the code base by preserving the behaviour of classes and the interfaces they implement) and renaming of classes and interfaces. It might be useful to perform SPL evolution that requires changing existing artifacts by performing a lightweight evolution followed by a refactoring and a renaming.

6. Related Work

The literature related to our proposal has been partially quoted through the paper. We add here further comparisons and remarks concerning programming languages with traits and programming languages for software product lines.

6.1. Programming Languages with Traits

Traits are well suited for designing libraries and enable clean design and reuse (as shown, e.g., Black et al. [16] and Cassu et al. [21]). Recently, Bergel et al. [9] pointed out limitations of the original trait model [62, 29] (methods provided by a trait can only access state by accessor methods) and propose stateful traits by adding private fields that can be accessed from the clients possibly under a new name or merged with other variables. In FPTJ traits are stateless, however, it is possible to directly access state within the methods provided by a trait by their required fields. Moreover, the names of required fields (in traits) can be changed in an unanticipated way by means of the field rename operation. Since field renaming works synergistically with the trait parameterization mechanism and with method renaming, exclusion and aliasing, FPTJ traits have more reuse potential than stateful traits.

FPTJ requires that the summed traits must be disjoint. The disjoint requirement for composed unit of reuse was proposed by Snyder [66] for multiple class-based inheritance (see also Bracha’s JIGSAW framework [19]). According to other proposals, two methods with the same name do not conflict if they are syntactically equal (Ducasse et al. [29, 51]) or if they originate from the same subtrait (Liquori and Spiwack [45]). In FPTJ, when a recursive method is aliased its recursive invocation refers to the original method (as proposed by Schärlí et al. [62, 29]). The variant of aliasing proposed by Liquori and Spiwack [45] (where, when a recursive method is aliased, its recursive invocation refers to the new method) can be straightforwardly encoded by exclusion, renaming and symmetric sum. Instead, exclusion, renaming, symmetric sum and the variant of aliasing are not able to encode aliasing. Concerning method renaming and required field renaming, they are not present in most formulations of traits in the SMALLTALK/SQUEAK-like and JAVA-like settings. Method renaming has been introduced in the formulation of traits in a structurally typed setting by Reppy and Turon [57]. Renaming operations were already present in the JIGSAW framework [19] in connection with module composition (more recently, Lagorio et al. [40, 42] defined an instantiation of the JIGSAW framework within a JAVA-like nominal type system) and in the EIFFEL language [48] in connection with multiple class-based inheritance.

Reppy and Turon [58] proposed a variant of traits that can be parametrized by member names (field and methods), types and values. In their proposal, the programmer can write trait functions that can be seen as code templates to

\(^{10}\)The use closure of each class can be computed once, since it does not change until one of its members is updated.
be instantiated with different parameters. This mechanism (termed \textit{trait-based metaprogramming}) enhances the code reuse provided by traits already. An important difference between our notion of parametric trait and the one by Reppy and Turon [58] is that, in the latter, trait parameters have a static scoping. Instead, in FPTJ trait parameters have a dynamic scoping and can be introduced in an unanticipated way.

\textsc{Scala} [52] provides a mixin construct (termed “trait” in the \textsc{Scala} syntax) and employs an emulated version of \textit{deep mixin composition} (a mechanism introduced by Ernst in [30]) as the main mechanism to express scalable extensibility (see, e.g., [46, 53]). Deep mixin composition makes it possible to express a version of the expression problem including two-dimensional extension and merging [31]. The language integrated version of deep mixin composition [30, 31] handles name clashes automatically (allowing class composition to take place even at run-time), whereas our approach requires programmer intervention in order to handle name clashes and similar issues and in return provides greater flexibility, including the ability to remove declarations.

6.2. Programming Languages for Software Product Lines

Standard class-based inheritance allows code reuse only within the class hierarchy and thus it is often too restrictive to implement feature-based variability of SPLs. Furthermore, inheritance does not support the removal of product functionality. Hence, there are several approaches providing other linguistic constructs for flexibly implementing the variability of SPLs in the object-oriented paradigm. The approaches to implementing SPLs can be classified into two main directions [38]. First, \textit{annotative approaches}, such as conditional compilation, frames [6] and \textsc{Colored Featherweight Java} (CFJ) [36], mark the source code of the whole SPL with respect to product features and remove marked code depending on the feature configuration. Second, \textit{compositional approaches} (like the calculus FPTJ presented in this paper) assemble products from artifacts in a common artifact base.

Compositional implementations of SPLs in the object-oriented paradigm use a variety of program modularization mechanisms, such as aspects [37], framed aspects [47], mixins [64], or hyperslices [68]. In these approaches, feature-based variability is restricted to the expressivity of the underlying programming paradigm. In [46], product line variability is implemented in \textsc{Scala} [52] using mixin-based inheritance. While \textsc{Scala} provides means to modularize classes and to extend them by adding classes, fields and methods via mixins (called “traits” in \textsc{Scala}), the specification of the desired composition is less flexible than in FPTJ. Most of these approaches do not have first-class operations to remove code which, however, are necessary to capture SPL evolution, as pointed out in this paper.

In feature-oriented programming (FOP) [8], the implementation of a product line is modularized into feature modules, each referring to one product feature. Feature modules can define new classes and refine existing classes. In order to realize a particular feature configuration, the respective feature modules are composed. The calculus \textsc{Lightweight Feature Java} (LFJ) [28], based on LJ (\textsc{Lightweight Java}) [67] provides a formalization of FOP together with a constraint-based type system (similar to the one in [2]) that supports the type-checking of feature modules in isolation. For each feature module, a set of constraints is inferred that are imposed by the introduction and refinement operations of the feature modules. The type safety of a SPL in LFJ can be verified by checking the validity of a generated propositional formula expressing the type safety of all products that can be derived according to the constraints of the feature model. The \textsc{Featherweight Feature Java} for Product Lines (FFJ\textsubscript{PL}) calculus [4] proposes an independently developed type checking approach for feature-oriented product lines. FFJ\textsubscript{PL} relies on FFJ [5], a calculus for stepwise-refinement, that is not explicitly bound to implementing SPLs. In FFJ\textsubscript{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 \textsc{Java} code. The FFJ\textsubscript{PL} typing rules do not generate constraints, but directly consult the feature model. Modular type-checking is not supported in FFJ\textsubscript{PL} since each feature module is analyzed by relying on information of the complete product line.

Delta-oriented programming (DOP) [59, 61] is an extension of feature-oriented programming. The implementation of a SPL in DOP is split into delta modules which extend feature modules by including removal of classes, methods and fields. A particular product is generated by applying the modifications of the applicable delta modules in an order that is compatible with an explicitly specified application ordering. In [14], a compositional type system for IFAJ, a core calculus for delta-oriented product lines of \textsc{Java} programs based on IFJ (\textsc{Imperative Featherweight Java}) is presented. Similar to LFJ, it is equipped with a constraint-based type system that infers constraints for each delta module in isolation. In LFJ [28] and IFAJ [14], a SPL with new products can be type-checked by analyzing only the code of the newly added feature or delta modules (these type systems have similarities with [2]). FPTJ adopts a similar technique. If new artifacts are added to the artifact base, the existing classes, traits and interface
do not have to be reanalyzed: it suffices to type-check the code of the new classes, traits and interfaces. The main
difference between the FPTJ approach for implementing SPL and the approaches based on FOP and DOP is that with
FPTJ the classes, interfaces and parametric traits of all the products coexist in the artifact base. Generation of a single
product just amounts to selecting a use-closed subset of these artifacts. Therefore, a class/interface/parametric-trait
name is associated to the same definition entity in all the products. This makes the approach particularly suitable for
supporting the lightweight form of SPL evolution outlined at the end of Section 5.

7. Conclusions and Future Work

In this paper, we presented a novel approach to implement product line variability by parametric traits. The
FPTJ type system is able to ensure type-safety of a SPL by type-checking its artifacts only once and to ensure type-
safety of an extension of a (type-safe) SPL by checking only the newly added parts. For future work, we plan to
develop a prototypical implementation of a language based on the FPTJ calculus (a prototypical implementation of
TRAITRECORDJ [15] and a more complete implementation of XTRAITJ [11], which are languages based on variants
of the calculus that do not include the trait parameterization mechanism, are available at http://traitrecordj.sf.net
and http://xtraitj.sf.net, respectively). Additionally, we aim at developing a process and guidelines for building
up an artifact base supporting as much code reuse as possible for implementing a particular SPL. To support this
process, we will develop an IDE to view the different code artifacts from the perspective of the product line declaration
and to manage lightweight SPL evolution and refactoring of classes and interfaces which is essential in order to make
the proposed approach scalable. Recently, deductive proof system for verifying behavioral properties of trait-based
programs [63, 27, 25] and for verifying behavioral properties of software product lines [33, 70, 26, 25] have been
investigated. We believe that a transformational approach similar to [26] could be applied to parametric traits and to
the verification of software product lines implemented by the approach presented in this paper.

Acknowledgments. We are grateful to Alexandre Bergel, Viviana Bono, Damien Cassou, Stéphane Ducasse, Marco
Servetto and Elena Zucca for many interesting discussions on traits and thank the anonymous OOPS 2010 and SCP
referees for insightful comments and suggestions for improving the presentation.

A. FFPTJ Type Soundness

In the following we will write “# ı” to denote length of a sequence ı.

A.1. FFPTJ 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 I. Values, ranged over by the metavariable
v are either addresses or null. Objects are denoted by ⟨C, f = v⟩, where C is the class of the object, f are the names of
the fields and v are the values of the fields. A heap H is a mapping from addresses to objects. The empty heap will
be denoted by / 0. 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 H, e. The reduction relation has the form H, e −→ H′, e′, to read “the
configuration H, e reduces to the configuration H′, e′ in one step”. The initial configuration associated to a program
(IT, •, CT), such that

\[
\begin{align*}
IT(\text{IMain}) &= \text{interface IMain extends} \bullet \{ \text{main}(); \} \text{ and} \\
CT(\text{CMain}) &= \text{class CMain implements IMain by} \{ \bullet: \\text{main}() \{ \text{return e;} \} \} \{ \bullet \}
\end{align*}
\]

is / 0, e.

The reduction rules shown in Figure 12, by 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 fields, which was defined in Figure 6, and mbody (relying
on the auxiliary function methods defined in Figure 6).
Computation rules:

\[
\mathcal{H}(t) = \langle C, \bar{f} = \bar{v} \rangle
\]
\[
\mathcal{H}, t. f_i \rightarrow \mathcal{H}, v_i \quad \text{(R-FIELD)}
\]

\[
\mathcal{H}(t) = \langle C, \ldots \rangle \quad \text{mbody}(m, C) = (\bar{x}, e_0)
\]
\[
\mathcal{H}, t. m(\bar{v}) \rightarrow \mathcal{H}, [\bar{v} \leftarrow \bar{v}, \text{this} \leftarrow t] e_0 \quad \text{(R-INV)}
\]

\[
t \notin \text{dom}(\mathcal{H}) \quad \text{fields}(C) = \bar{f}\bar{i}
\]
\[
\mathcal{H}, \text{new C()} \rightarrow \mathcal{H} \cup \{t \mapsto \langle C, \bar{f} = \text{null} \rangle\}, t \quad \text{(R-NEW)}
\]

\[
\mathcal{H}(t) = \langle C, \bar{f} = \bar{v} \rangle
\]
\[
\mathcal{H}, t. f_i = v \rightarrow \mathcal{H} [t \mapsto \langle C, \ldots, f_i = v \ldots \rangle], v \quad \text{(R-ASSIGN)}
\]

\[
\mathcal{H}(t) = \langle C, \bar{f} = \bar{v} \rangle \quad C <: \text{I}
\]
\[
\mathcal{H}, (I) t \rightarrow \mathcal{H}, t \quad \text{(R-CAST)}
\]

\[
\mathcal{H}, (I) \text{null} \rightarrow \mathcal{H}, \text{null} \quad \text{(R-CASTN)}
\]

Congruence rules:

\[
\mathcal{H}, e \rightarrow \mathcal{H}', e'
\]
\[
\mathcal{H}, e. m(\bar{v}) \rightarrow \mathcal{H}', e. m(\bar{v})
\]
\[
\mathcal{H}, e \rightarrow \mathcal{H}', e'
\]
\[
\mathcal{H}, t. f_i = e \rightarrow \mathcal{H}, t. f_i = e'
\]
\[
\mathcal{H}, (I) e \rightarrow \mathcal{H}', (I) e'
\]

Method body lookup:

\[
\text{methods}(C)(m) = \text{Im}(\bar{m})(\bar{x}) \{ \text{return } e; \}
\]
\[
mbody(m, C) = (\bar{x}, e)
\]

Figure 12: FFPTJ: Operational semantics
A.2. FFPTJ Type Soundness

In order to be able to formulate the type soundness of FFPTJ as a subject reduction 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 [34]), i.e., a cast where the subject and the target are unrelated, or a stupid selection, i.e., a method invocation null.m(···), are not well typed according to the FFPTJ (source level) type system\(^{11}\). However, a runtime expression without stupid casts and stupid selections may reduce to a runtime expression containing either a stupid cast or a stupid 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 Figure 13; these rules use the environment \(\Sigma\), which is a finite (possibly empty) mapping from addresses to class names, and they are of the shape \(\Sigma \vdash \tau : E\). In Figure 13 we also present the notion of well-formed heap and of well-formed configuration. The notion of well-formed heap ensures that the environment \(\Sigma\) 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 A.1.** If \(I \vdash m(\vec{I}) \in mSig(methods(\mathcal{C}))\) and \(mbody(\mathcal{C}, m) = (\vec{x}, e)\) then we have \(\text{this} : \mathcal{C}, \vec{x} : \vec{I} \vdash_1 e : E\) for some \(E \ll I\).  

**PROOF.** Follows directly from the definition of \(mbody\), \(mSig\) and rule (T-METHOD). \(\Box\)

**Lemma A.2 (Substitution).** If
1. \(\Sigma \vdash \tau_1, m(\vec{y}) : I\) where \(\Sigma(\tau) = \mathcal{C}\) for some \(\Sigma, \mathcal{C}\) and \(I\),
2. \(I \vdash m(\vec{I}) \in mSig(methods(\mathcal{C}))\), and
3. \(mbody(\mathcal{C}, m) = (\vec{x}, e)\),
then we have \(\Sigma \vdash \vec{x} \leftarrow \vec{v}, \text{this} \leftarrow t[\vec{x}] : E\) for some \(E \ll \vec{I}\).

**PROOF.** By hypotheses 1. and 2. and by Lemma A.1, for some \(E \ll I\), we have \(\text{this} : \mathcal{C}, \vec{x} : \vec{I} \vdash_1 e : E\).

The proof then proceeds by structural induction on the derivation of \(\text{this} : \mathcal{C}, \vec{x} : \vec{I} \vdash_1 e : E\). We present only a few interesting cases (the cases for casts are the same as in FJ, in particular, for (T-NCAST) we can use (RT-SCAST)). Note that, by rule (RT-INV), \(\Sigma \vdash \vec{v} : \vec{E}\) for some \(\vec{E}\) such that \(\vec{E} \ll \vec{I}\) (in particular, \(E_I = I_I\) when \(v_i = \text{null}\) by rule (RT-NUL)).

**Case (T-VAR)** In this case \(e = x_i\) for some \(x_i \in \vec{x}\): \([\vec{x} \leftarrow \vec{v}, \text{this} \leftarrow t[\vec{x}] = v_i, \Sigma \vdash \tau_1 v_i : E_i\) for some \(E_i\) such that \(E_i \ll I_i\); letting \(E_I = E\) finishes the case.

**Case (T-FIELD)** In this case \(e = \text{this}.f\). By rule (T-FIELD) we have \(\text{this} : \mathcal{C}, \vec{x} : \vec{I} \vdash_1 \text{this} : \mathcal{C}\) and \(f \in \text{fields}(\mathcal{C})\), where \(J = E\). By hypothesis 1. we have \(\Sigma(\tau) = \mathcal{C}\). The thesis follows by applying (RT-FIELD) to \([\vec{x} \leftarrow \vec{v}, \text{this} \leftarrow t[\vec{x}] = t.f\) obtaining \(\Sigma \vdash_1 t.f : J\).

**Case (T-INV1)** In this case \(e = \text{this}.m'(\vec{a})\). We proceed as in the previous case (using (RT-INV)).

**Case (T-INV2)** In this case \(e = e'.m'(\vec{a})\) (where \(e' \neq \text{this}\)). By (T-INV2) we have

\[
\begin{align*}
\Gamma \vdash e' : N' & \quad \quad \quad m'(\vec{J}) \in mSig(N') & \quad \quad \quad \Gamma \vdash e : \vec{E} & \quad \quad \quad \vec{E} \ll \vec{J} \\
\end{align*}
\]

where \(J = E \ll I\). Using the induction hypothesis on \(e'\) and \(\vec{a}\), we have \(\Sigma \vdash_1 [\vec{x} \leftarrow \vec{v}, \text{this} \leftarrow t[\vec{x}] : N' \ll \vec{N'}\) \(\Sigma \vdash_1 \vec{x} \leftarrow \vec{v}, \text{this} \leftarrow t[\vec{x}] : \vec{E} \ll \vec{E} = \vec{E}'\), using the fact that \(\text{choose}(mSig(N'), \vec{m'}) = \text{choose}(mSig(methods(N')), \vec{m'})\) (if \(N'\) is a class name) and \(\text{choose}(mSig(N'), \vec{m'}) = \text{choose}(mSig(N'), \vec{m'})\) (otherwise) we can use (RT-INV) to obtain the thesis.

\(^{11}\)Note that expressions such as \(\text{null}.f\) are prevented by the syntax itself.
Runtime expression typing \[ \Sigma \vdash r : T \]

\[ \Sigma \vdash t : \Sigma(t) \]

\[ \Sigma \vdash t : C \quad \text{if } t \in \text{fields}(C) \]

\[ \Sigma \vdash r : N \]

\[ \Sigma \vdash r_\bar{e} : \bar{E} \quad \bar{E} < : I \]

\[ \Sigma \vdash r : C \quad \text{if } C \in \text{fields}(C) \]

\[ \Sigma \vdash r : I \quad \text{I m}(I) \in \{ \text{mSig(methods}(C)) \quad \text{if } C \text{ is a class name} \]

\[ \Sigma \vdash r : C \quad \text{if } C \notin \text{fields}(C) \]

\[ \Sigma \vdash r : I \quad I \text{m}(I) \in \{ \text{mSig(methods}(C)) \quad \text{if } C \text{ is a class name} \]

\[ \Sigma \vdash r : C \quad C < : I \quad \text{stupid warning} \]

\[ \Sigma \vdash r : E \quad E < : I \]

\[ \Sigma \vdash t.f : I \]

\[ \Sigma \vdash \text{null} : E \quad E \in \{ \bot \} \cup \text{dom}(IT) \cup \text{dom}(CT) \]

Well-formed heap \[ \Sigma \vdash H \]

\[ \text{dom}(H) = \text{dom}(\Sigma) \]

\[ \forall t \in \text{dom}(H), \ H(t) = (C, f_1 = v_1, \ldots, f_n = v_n) \quad \text{implies} \quad \left( \begin{array}{c} \Sigma(t) = C \\ \text{fields}(C) = I_1 f_1, \ldots, I_n f_n \\ \forall i \in 1..n, \ \Sigma \vdash r : E_i \quad E_i < : I_i \end{array} \right) \]

Well-typed configuration \[ \Sigma \vdash H, e : T \]

\[ \Sigma \vdash e : E \quad \Sigma \vdash H \quad \Sigma \vdash e : E \]

\[ \Sigma \vdash H, e : E \]
Case (T-ASSIG) The thesis follows from the induction hypothesis and the transitivity of $\prec$. □

Lemma A.3 (Weakening). If $\Sigma \vdash_{\mathcal{R}} e : \mathcal{E}$ then $\Sigma, t : \mathcal{C} \vdash_{\mathcal{R}} e : \mathcal{E}$.

PROOF. Straightforward induction on the derivation of $\Sigma \vdash_{\mathcal{R}} e : \mathcal{T}$. □

Theorem A.4 (Subject reduction). If $\Sigma \models \mathcal{H}, \Sigma \vdash_{\mathcal{R}} e : \mathcal{E}$ and $\mathcal{H}, e \longrightarrow \mathcal{H}', e'$ then there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma' \models \mathcal{H}', \Sigma' \vdash_{\mathcal{R}} e' : \mathcal{E}'$ for some $\mathcal{E}' \prec \mathcal{E}$.

PROOF. The proof is by induction on a derivation of $\mathcal{H}, e \longrightarrow \mathcal{H}', e'$, 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 A.3).

Case (R-FIELD) The last applied rule is $\mathcal{H}, i{:}f \longrightarrow \mathcal{H}, v$, where $\mathcal{H}(i) = \langle \mathcal{C}, \hat{f} = \bar{v} \rangle$. By hypothesis $\Sigma \vdash_{\mathcal{R}} i{:}f : I_i$ and by (WF-HEAD) we have $\Sigma \vdash_{\mathcal{R}} v : \mathcal{E}_i$ for some $\mathcal{E}_i \prec I_i$. Thus we have the thesis.

Case (R-INVK) The last applied rule is

\[
\mathcal{H}(i) = \langle \mathcal{C}, \hat{f} = \bar{v} \rangle \quad \text{mbody}(m, \mathcal{C}) = (\bar{x}, e_0) \\
\mathcal{H}, i{:}m(\bar{v}) \longrightarrow \mathcal{H}, \overline{x} \leftarrow \bar{v}, \text{this} \leftarrow i[e_0]
\]

Since by hypothesis $\Sigma \vdash_{\mathcal{R}} i{:}m(\bar{v}) : I$ the thesis follows by applying Lemma A.2.

Case (R-NEW) Let $\Sigma' = \Sigma \cup \{ i : \mathcal{C} \}$. By hypothesis $\Sigma \models \mathcal{H}$, and by applying (WF-HEAD) we also have $\Sigma' \models \mathcal{H} \cup \{ i \mapsto (\mathcal{C}, \hat{f} = \text{null}) \}$. $\Sigma' \vdash_{\mathcal{R}} i : \mathcal{C}$ follows from (RT-ADD).

Case (R-ASSIGN) By rule (T-ASSIG) we have that $\Sigma \vdash_{\mathcal{R}} v : \mathcal{E}'$ and $\mathcal{E}' \prec \mathcal{E}$ for some $\mathcal{E}'$. By hypothesis $\Sigma \models \mathcal{H}$, and by applying (WF-HEAD) we also have $\Sigma \models \mathcal{H}[\mathcal{H}(i) \mapsto (\mathcal{C}, \ldots, f_i = v, \ldots)]$.

Lemma A.5. Let $\mathcal{H}, e$ be a well-typed configuration.

1. If $e = i{:}f$ then $\mathcal{H}(i) = \langle \mathcal{C}, \ldots \rangle$ and $f \in \text{fields}(\mathcal{C})$ for some $I$.
2. If $e = i{:}m(\bar{v})$ then $\mathcal{H}(i) = \langle \mathcal{C}, \ldots \rangle$, $I_m(\bar{I}) \in \text{mSig}(\text{methods}(\mathcal{C}))$ and $\bar{z}(\bar{I}) = \bar{z}(\bar{e})$.

PROOF. Straightforward. □

We formulate the progress theorem in the same shape used for FJ in [54] (Theorem 19.5.4). Thus, we first introduce the notion of evaluation context for IFJ runtime expressions. The evaluation contexts $\mathcal{E}$ for IFJ runtime expressions are defined as follows:

\[
\mathcal{E} ::= \epsilon \mid \epsilon.f \mid \mathcal{E}.m(\bar{e}) \mid \nu.\mathcal{E}(\bar{v}, \bar{e}) \mid (\bar{I})\mathcal{E} \mid \epsilon.f = \mathcal{E}
\]

Theorem A.6 (Progress). Let $\mathcal{H}, e$ be a well-typed normal form. then

1. either $e$ is a value, or
2. for some evaluation context $\mathcal{E}$ we can express $e$ as
   (a) either $\mathcal{E}[(\bar{I})1]$ such that $\mathcal{H}(1) = \langle \mathcal{C}, \ldots \rangle$ with $\mathcal{C} \not\models I$, or
   (b) $\mathcal{E}[\text{null}.m(\bar{v})]$ for some $m$ and $\bar{v}$.

PROOF. Straightforward induction on typing derivations using Lemma A.5. □

Lemma A.7. If $\bullet \vdash_{\mathcal{T}} e : \mathcal{E}$ then $\bullet \vdash_{\mathcal{R}} e : \mathcal{E}$.

27
PROOF. Straightforward induction on typing derivations.

Theorem A.8 (Type Soundness). Consider program \((IT, \bullet, CT)\), such that

1. \(IT(IMain) = \text{interface } IMain\text{ extends } \{ IMain\} \) and
2. \(CT(CMain) = \text{class } CMain \text{ implements } IMain \text{ by } \{ \bullet; \bullet: IMain \{ \text{return } e; \} \} \{ \bullet \}\)

If \(\vdash_t CT \text{ OK}, \bullet \vdash_t e : E \text{ and } 0,0 \rightarrow^* H, e' \text{ with } H, e' \text{ a normal form. Then } e' \text{ is}

1. \(\text{either null,}\)
2. \(\text{or an address } t \text{ such that } H(t) = \langle C, \ldots \rangle \text{ with } C <: E,\)
3. \(\text{or for some evaluation context } \delta \text{ we can express } e' \text{ as}\)

   a. \(\delta[(I)t] \text{ such that } H(t) = \langle C, \ldots \rangle \text{ with } C \not\prec I, \text{ or}\)
   b. \(\delta[\text{null.m}(\overline{v})] \text{ for some } m \text{ and } \overline{v}.\)

PROOF. Follows from Lemma A.7, Theorem A.4 and Theorem A.6.

B. Proof of Theorem 4.2 (Flattening Preserves the Type of Programs)

Let \(\text{classNames}(TE)\) and \(\text{interfaceNames}(TE)\) denote the set of interface names and class names occurring in the trait expression \(TE\), respectively.

The sequence of the field names and the sequence of the method names selected on this in the expressions \(e\) are denoted by \(\text{fieldNames}(e)\) and \(\text{methodNames}(e)\), respectively. The sequence of the field names and the sequence of the method names selected on this in the method declaration \(MD = IMain(Ix)\text{\{return } e\text{\}}\) are given by \(\text{fieldNames}(MD) = \text{fieldNames}(e)\) and \(\text{methodNames}(MD) = \text{methodNames}(e)\). The definitions of \(\text{fieldNames}\) and \(\text{methodNames}\) naturally extend to sequences of expressions and sequences of method definitions.

Recall that the flattening \(\text{[TE]}_{CT}\) of a trait expression \(TE\) yields a sequence of methods (see Section 4.2). A sequence of methods \(\overline{MD}\) is well-typed if and only if all methods in \(\overline{MD}\) are well-typed. In the following, we will write “\(\langle \overline{F}D : \overline{\sigma} \rangle \vdash_t MD_1 \cdots MD_n : \mu_1 \cdots \mu_n\)” as short for “\(\langle \overline{F}D : \overline{\sigma}rg\rangle \vdash_t MD_1 : \mu_1, \ldots \langle \overline{F}D : \overline{\sigma}rg\rangle \vdash_t MD_n : \mu_n\)”.

Lemma B.1. If \(\langle \overline{F}D : \overline{\sigma}rg\rangle \vdash_t MD : \zeta \vdash \langle \overline{F}D' : \overline{\sigma}'rg\rangle \vdash \zeta'\), then \(\langle \overline{F}D'' : \overline{\sigma}''rg\rangle \vdash_t MD : \zeta \vdash \langle \overline{F}D' : \overline{\sigma}'rg\rangle \vdash \zeta'\) \(\text{for all } \overline{F}D'' \supseteq \overline{F}D' \text{ and } \overline{\sigma}'' \supseteq \overline{\sigma}'\).

PROOF. By structural induction on typing derivations.

Lemma B.2. If \(\langle \overline{F}D : \overline{\sigma}rg\rangle \vdash_t MD : \mu, \text{classNames}(MD) = \xi, \text{interfaceNames}(MD) = \eta, \text{then for all set of classes } \overline{BD} \text{ and set of interfaces } \overline{BI}\) such that \#\(D = \#\xi\) and \#\(I = \#\eta\) it holds that \(\langle \overline{F}D' : \overline{\sigma}'rg\rangle \vdash_t MD[\overline{B}D/\overline{C}I] : \mu[\overline{B}D/\overline{C}I]\).

PROOF. By structural induction on typing derivations.

Lemma B.3. Let \(\vdash_t TE : \mu_1 \cdots \mu_n, \text{where, for all } i \in 1..n, \mu_i = \zeta_i : \langle \overline{F}D(i) : \overline{\sigma}(i) \rangle \vdash \zeta_i\). Then \(\langle \overline{F}D : \overline{\sigma} \rangle \vdash_t [TE]_{CT} : \mu_1 \cdots \mu_n\), where \(\overline{F}D = \overline{F}D(1) \cup \cdots \cup \overline{F}D(n)\) and \(\overline{\sigma} = \overline{\sigma}(1) \cup \cdots \cup \overline{\sigma}(n)\).

PROOF. By case induction on the flattening translation for trait expressions defined in Figure 3.

Case \([\{ \overline{F}D, \overline{HI}, \overline{RI} \}]_{CT}\). This is the base case of the induction. Straightforward by rule \(\text{[CT-TEBASIC]}\) in Figure 9.

Case \([T(D)]_{CT}\). Straightforward by induction, using Lemma B.2.
Case $\llbracket \text{TE}_1 + \text{TE}_2 \rrbracket_{\text{TT}}$. Recall that $\llbracket \text{TE}_1 + \text{TE}_2 \rrbracket_{\text{TT}} = \llbracket \text{TE}_1 \rrbracket_{\text{TT}} + \llbracket \text{TE}_2 \rrbracket_{\text{TT}}$. By induction we have that $(\text{FD} \cdot \sigma') \vdash_{\text{ct}} \llbracket \text{TE}_1 \rrbracket_{\text{TT}} : \mu \cdot \bar{\sigma}$ and $(\text{FD} \cdot \bar{\sigma'}) \vdash_{\text{ct}} \llbracket \text{TE}_2 \rrbracket_{\text{TT}} : \mu''$, where $\text{FD} \subseteq \text{FD}, \text{FD} \subseteq \text{FD}, \sigma \subseteq \bar{\sigma}, \sigma' \subseteq \sigma$, and $\mu' \cdot \bar{\sigma'} = \mu_1 \cdots \mu_n$. Since, for all $i \in 1..n$, both $\text{FD} \supseteq \text{FD}^{(i)}$ and $\bar{\sigma} \supseteq \bar{\sigma}^{(i)}$ hold, then the result follows by Lemma B.1.

Case $\llbracket \text{TE}_0[\text{exclude m}] \rrbracket_{\text{TT}}$. Recall that $\llbracket \text{TE}_0[\text{exclude m}] \rrbracket_{\text{TT}} = \text{exclude}(\llbracket \text{TE}_0 \rrbracket_{\text{TT}}, m)$. By induction we have that $(\text{FD} \cdot \sigma') \vdash_{\text{ct}} \llbracket \text{TE}_0 \rrbracket_{\text{TT}} : \mu_1 \cdot \cdots \cdot \mu_n \cdot \bar{\sigma}$, where $\mu_0 = I_0 \llbracket (\bar{\sigma}^{(0)}) \rrbracket (\text{FD}^{(0)} \cdot \bar{\sigma}^{(0)}) \cdot \zeta_0$, $\text{FD} = \text{FD} \cup \text{FD}^{(0)}$ and $\sigma' = \sigma \cup I_0 \cup \mu_0 \cup \mu_n \cup \bar{\sigma}^{(0)}$. Since, for all $i \in 1..n$, both $\text{FD} \supseteq \text{FD}^{(i)}$ and $\bar{\sigma} \supseteq \bar{\sigma}^{(i)}$ hold, then the result follows by Lemma B.1.

Case $\llbracket \text{TE}_0[\text{alias As m}'] \rrbracket_{\text{TT}}$. Recall that $\llbracket \text{TE}_0[\text{alias As m}'] \rrbracket_{\text{TT}} = \llbracket \text{TE}_0 \rrbracket_{\text{TT}} \cdot (I_p \llbracket (\bar{\sigma}^{(p)}) \rrbracket (\text{return e}_p)))$, where $I_p \llbracket (\bar{\sigma}^{(p)}) \rrbracket (\text{return e}_p); \cdot \mu_p, \text{then the fact that } (\text{FD} \cdot \sigma') \vdash_{\text{ct}} I_p \llbracket (\bar{\sigma}^{(p)}) \rrbracket (\text{return e}_p); \cdot \mu_p$ holds.

$m' \notin \bar{\sigma}$. We have $\bar{\sigma} = \sigma' \cdot (I_p \llbracket (\bar{\sigma}^{(p)}) \rrbracket (\text{return e}_p))).$

- Since, for all $i \in 1..n - 1$, $\bar{\sigma} \supseteq \bar{\sigma}^{(i)}$ holds, then (by Lemma B.1) we have that $(\text{FD} \cdot \sigma') \vdash_{\text{ct}} \llbracket \text{TE}_0 \rrbracket_{\text{TT}} : \mu_1 \cdots \mu_{n-1}$.
- Since $(\text{FD} \cdot \bar{\sigma'}) \vdash_{\text{ct}} I_p \llbracket (\bar{\sigma}^{(p)}) \rrbracket (\text{return e}_p); : \mu_p$, then the fact that $(\text{FD} \cdot \bar{\sigma'}) \vdash_{\text{ct}} I_p \llbracket (\bar{\sigma}^{(p)}) \rrbracket (\text{return e}_p); : \mu_p$ holds.

$m' \in \bar{\sigma}$. Note that this case can happen only if the method $m$ (provided by $\text{TE}_0$) and the method $m'$ (required by $\text{TE}_0$) have the same signature, otherwise $\llbracket \text{TE}_0[\text{renameTo m}'] \rrbracket_{\text{TT}}$ would have not been well-typed, which contradicts the hypothesis. We have $\bar{\sigma} = \sigma'$, therefore the only thing that remains to be proved is that $(\text{FD} \cdot \bar{\sigma'}) \vdash_{\text{ct}} I_p \llbracket (\bar{\sigma}^{(p)}) \rrbracket (\text{return e}_p); : \mu_p$.

Case $\llbracket \text{TE}_0[\text{renameTo m}] \rrbracket_{\text{TT}}$. Recall that $\llbracket \text{TE}_0[\text{renameTo m}] \rrbracket_{\text{TT}} = mR(\llbracket \text{TE}_0 \rrbracket_{\text{TT}}, m, m')$. By induction we have that $(\text{FD} \cup \text{FD}^{(p)}) \vdash_{\text{ct}} \llbracket \text{TE}_0 \rrbracket_{\text{TT}} : \mu_1' \cdots \mu_n' \cdot \bar{\sigma}^{(i)}$, where $\mu_1' = \zeta_i' \cdot \llbracket (\text{FD}^{(p)} \cdot \bar{\sigma}^{(i)}) \rrbracket \cdot \zeta_i$ (for all $i \in 1..n$, $m \in \text{names}(\bar{\sigma})$ and $m' \notin \text{names}(\mu_1' \cdots \mu_n')$. We consider four different cases.

$m' \notin \bar{\sigma}$. We have $\bar{\sigma} = \bar{\sigma}' \cdot [m/m']$ and, for all $i \in 1..n$, $\zeta_i = \zeta_i'[m/m']$ and $\bar{\sigma}^{(i)} = \bar{\sigma}^{(i)}[m/m']$.

$m' \in \bar{\sigma}'$ and $m \notin \text{names}(\mu_1' \cdots \mu_n')$. Note that this case can happen only if the method $m$ (provided by $\text{TE}_0$) and the method $m'$ (required by $\text{TE}_0$) have the same signature, otherwise $\llbracket \text{TE}_0[\text{renameTo m}'] \rrbracket_{\text{TT}}$ would have not been well-typed, which contradicts the hypothesis. We have $\bar{\sigma} = \text{exclude}(\bar{\sigma}', m)$ and, for all $i \in 1..n$, $\zeta_i = \zeta_i'[m/m']$ and $\bar{\sigma}^{(i)} = \text{exclude}(\bar{\sigma}^{(i)}[m/m'])$.

$m' \in \bar{\sigma}'$, $m \notin \text{names}(\mu_1' \cdots \mu_n')$ and $m' \notin \text{names}(\mu_1' \cdots \mu_n')$. Note that this case can happen only if the method $m$ (required by $\text{TE}_0$) and the method $m'$ (provided by $\text{TE}_0$) have the same signature, otherwise $\llbracket \text{TE}_0[\text{renameTo m}'] \rrbracket_{\text{TT}}$ would have not been well-typed, which contradicts the hypothesis. We have $\bar{\sigma} = \text{exclude}(\bar{\sigma}') \cdot m \cup (\text{choose}(\bar{\sigma}^{(i)}[m/m']))$.

$m' \in \bar{\sigma}'$, $m \notin \text{names}(\mu_1' \cdots \mu_n')$ and $m' \notin \text{names}(\mu_1' \cdots \mu_n')$. Note that this case can happen only if the method $m$ (required by $\text{TE}_0$) and the method $m'$ (required by $\text{TE}_0$) have the same signature, otherwise $\llbracket \text{TE}_0[\text{renameTo m}'] \rrbracket_{\text{TT}}$ would have not been well-typed, which contradicts the hypothesis. We have $\bar{\sigma} = \text{exclude}(\bar{\sigma}') \cdot m \cup (\text{choose}(\bar{\sigma}^{(i)}[m/m']))$.

In all the four cases the result can be proved straightforwardly by structural induction on typing derivations.

Case $\llbracket \text{TE}_0[\text{renameTo f}] \rrbracket_{\text{TT}}$. Recall that $\llbracket \text{TE}_0[\text{renameTo f}] \rrbracket_{\text{TT}} = \llbracket \text{TE}_0 \rrbracket_{\text{TT}} [f'/f]$. By induction we have that $(\text{FD} \cdot \bar{\sigma}) \vdash_{\text{ct}} \llbracket \text{TE}_0 \rrbracket_{\text{TT}} : \mu_1' \cdots \mu_n'$ where $\mu_1' = \zeta_i' \cdot (\text{FD}^{(i)} \cdot \bar{\sigma}^{(i)}) \cdot \zeta_i$ (for all $i \in 1..n$) and $f \in \text{names}(\text{FD})$. We consider two different cases.

$f' \notin \bar{\sigma}'$. We have $\text{FD} = \text{FD}^{(p)}[f'/f]$ and, for all $i \in 1..n$, $\text{FD}^{(i)} \equiv \text{FD}^{(i)}[f'/f]$.
Lemma B.4. If \( \langle \mathcal{F}D; \sigma \rangle ; \) this : thisClass, \( x : I \vdash e : \varepsilon \mid \langle \mathcal{F}D'; \sigma' \rangle ; C \) holds with respect to \( P \), then it holds with respect to \( [P]_{\mathcal{T}T} \).

PROOF. Let \( P = (IT, TT, CT) \). Then \([P]_{\mathcal{T}T} = (IT, \bullet, [CT]_{\mathcal{T}T})\). The result is straightforward, since the constraint-based typing rules in Figure 11 do not use the trait table \( TT \) and the only rule that uses the class table \( CT \), rule \( (CT-new) \), does not distinguish between \( CT \) and \([CT]_{\mathcal{T}T}\).

Lemma B.5. If \( \langle \mathcal{F}D; \sigma \rangle \vdash \mathcal{M}D : \mu \) holds with respect to \( P \), then it holds with respect to \( [P]_{\mathcal{T}T} \).

PROOF. Straightforward by rule \( (CT-method) \) in Figure 9 and Lemma B.4.

Lemma B.6. If \( \vdash \mathcal{C}D \ OK \) holds with respect to \( P \), then \( \vdash [\mathcal{C}D]_{\mathcal{T}T} \ OK \) holds with respect to \( [P]_{\mathcal{T}T} \).

PROOF. Let \( \mathcal{C}D = \text{class C implements I by TE} \{ \mathcal{F}D \} \), then

\[
[\mathcal{C}D]_{\mathcal{T}T} = \text{class C implements I by} \{ \mathcal{F}D; \bullet; [TE]_{\mathcal{T}T} \} \{ \mathcal{F}D \}
\]

According to rule \( (CT-class) \) in Figure 9, \( \vdash \mathcal{C}D \ OK \) holds w.r.t. to \( P \) implies \( \vdash \text{TE} : \mu_1 ... \mu_p \) holds w.r.t. to \( P \), where

\[
\forall i \in 1..p, \quad \mu_i = \zeta_i \mid (\mathcal{F}D^{(i)}; \sigma^{(i)}); \varepsilon \mid \varepsilon \vdash \mathcal{C}D_i ; \mathcal{C}D_i \{thisClass\} \ OK
\]

and \( \mathcal{F}D = \bigcup_{i \in 1..p} \mathcal{F}D^{(i)} \) and \( \bigcup_{i \in 1..p} \mathcal{F}D^{(i)} \cup mSig(I) \leq \zeta_1 ... \zeta_p \).

By Lemma B.3, we have \( \langle \mathcal{F}D; \zeta_1 \rangle \vdash [TE]_{\mathcal{T}T} : \mu_1 ... \mu_p \) holds w.r.t. to \( P \), where \( \mathcal{F}D = \mathcal{F}D^{(1)} \cup ... \cup \mathcal{F}D^{(p)} \) and \( \zeta = \zeta_1 ... \zeta_p \). By Lemma B.5 we have that \( \langle \mathcal{F}D; \zeta \rangle \vdash [TE]_{\mathcal{T}T} : \mu_1 ... \mu_p \) holds also with respect to \( [P] \). Therefore, by rule \( (CT-terBasic) \), \( \vdash \{ \mathcal{F}D; \bullet; [TE]_{\mathcal{T}T} \} : \mu_1 ... \mu_p \) holds w.r.t. \( [P] \) and, by rule \( (CT-class) \), \( \vdash [\mathcal{C}D]_{\mathcal{T}T} \ OK \) holds w.r.t. \( [P] \).

PROOF OF THEOREM 4.2 (Flattening Preserves the Type of Programs). Straightforward by Lemma B.6 and rule \( (CT-program) \) in Figure 9.

C. Proof of Theorem 4.3 (Equivalence of \( \vdash \mathcal{C}D \) -typability and \( \vdash e \) -typability on FFPTJ programs)

Lemma C.1. For every class definition class \( C \) implements \( I \) by \( \{ \mathcal{F}D; \bullet; \mathcal{M}D \} \{ \mathcal{F}D; \} \) with \( \sigma = mSig(\mathcal{M}D) \) it holds that: this : \( C; \check{x} : I \vdash e : \varepsilon \) if and only if

1. \( \langle \mathcal{F}D; \sigma \rangle ; \text{this : thisClass,} \check{x} : I \vdash e : \varepsilon \mid \langle \mathcal{F}D; \sigma' \rangle \mid \varepsilon, \) and
2. \( \vdash \varepsilon \mid \mathcal{C}D_i ; \mathcal{C}D_i \{thisClass\} ; S \) and \( E = \mathcal{E} \{thisClass\} \).

PROOF. By structural induction on typing derivations, using the constraint satisfaction checking rules. We show only the cases for the “only if” direction. The cases for the “if” direction are similar. Assume \( \text{this :} C; \check{x} : I \vdash e : \varepsilon \).

Case \( (T-var) \). Then \( e = x \) and \( \Gamma \vdash t : \Gamma(x) \). The result follows by rule \( (CT-var) \).

Case \( (T-field) \). Then \( e = \text{this}.f \) and \( \Gamma \vdash \text{this}.f : I \). The result follows by rule \( (CT-field) \).
Case (**T-new**). Then \( e = \text{new} C() \) and \( \Gamma \vdash t \text{new} C() : C \). The result follows by rule (**ct-new**).

Case (**T-null**). Then \( e = \text{null} \) and \( \Gamma \vdash t \text{null} : \bot \). The result follows by rule (**ct-null**).

Case (**T-assg**). Then \( e = \text{this}.f = e_1, \Gamma \vdash t \text{this}.f = e_1 : I, \Gamma \vdash t \text{this}.f : I, \Gamma \vdash t e_1 : E_1 \) and \( E_1 < : I \).

By induction

- \( \langle \text{FD} : \sigma \rangle; \text{this}: \text{thisClass}, \bar{x} : \bar{e}_t \vdash t \text{this}.f : I \vdash (\text{If} : \bullet) : \bullet \).
- \( \langle \text{FD} : \sigma \rangle; \text{this}: \text{thisClass}, \bar{x} : \bar{e}_t e_1 : e_1 (\langle \text{FD} : \sigma' \rangle) : C, \vdash \sigma e_1 = [C/\text{thisClass}] : S \) and \( E_1 = (\bar{e}_1^0[\text{thisClass}])S \).

The result follows by rules (**ct-assg**) and (**cc-sub**).

Case (**T-inv1**). Then \( e = \text{this}.m(e_1, \ldots, e_n), \Gamma \vdash t \text{this}: C, \Gamma \vdash t (I_1, \ldots, I_n) \in mSig(methods(C)), \) and (for all \( i \in 1..n \) \( \Gamma \vdash t e_i : E_i \) and \( E_i < : I_i \). By induction

- (for all \( i \in 1..n \) \( \langle \text{FD} : \sigma \rangle; \Delta \vdash t e_i : e_i (\langle \text{FD} : \sigma(i) \rangle) : C, \vdash \sigma e_i = [C/\text{thisClass}] : S_i \) and \( E_i = (\bar{e}_i^0[\text{thisClass}])S_i \).
- choose \( (\sigma, m) = \text{Im}(I_1, \ldots, I_n) \).

Let \( C = \cup_{i \in 1..n} e_i \cup \{ \text{sub}(e_i, I_i) \} \). The result follows by rules (**ct-inv1**), (**cc-meth**) and (**cc-sub**).

Case (**T-inv2**). Then \( e = e_0.m(e_1, \ldots, e_n), \Gamma \vdash t e_0 : N_0, \Gamma \vdash t (I_1, \ldots, I_n) \in mSig(N_0), \) and (for all \( i \in 1..n \) \( \Gamma \vdash t e_i : E_i \) and \( E_i < : I_i \). By induction

- (for all \( i \in 1..n \) \( \langle \text{FD} : \sigma \rangle; \Delta \vdash t e_i : e_i (\langle \text{FD} : \sigma(i) \rangle) : C, \vdash \sigma e_i = [C/\text{thisClass}] : S_i \) and \( N_0 = (\bar{e}_i^0[\text{thisClass}])S_0 \).
- (for all \( i \in 1..n \) \( \langle \text{FD} : \sigma \rangle; \Delta \vdash t e_i : e_i (\langle \text{FD} : \sigma(i) \rangle) : C, \vdash \sigma e_i = [C/\text{thisClass}] : S_i \) and \( E_i = (\bar{e}_i^0[\text{thisClass}])S_i \).

Let \( C = (\cup_{i \in 1..n} e_i) \cup \{ \text{meth}(v_0, m, \alpha \alpha_1 \cdots \alpha_n) \} \cup (\cup_{i \in 1..n} \{ \text{sub}(e_i, I_i) \}) \). The result follows by rules (**ct-inv2**) and (**cc-sub**).

Case (**T-ucast**). Then \( e = (I)e_1, \Gamma \vdash t e_1 : E_1 \) and \( E_1 < : I \). By induction

- \( \langle \text{FD} : \sigma \rangle; \text{this}: \text{thisClass}, \bar{x} : \bar{e}_t e_1 : e_1 (\langle \text{FD} : \sigma' \rangle) : C, \vdash \sigma e_1 = [C/\text{thisClass}] : S \) and \( E_1 = (\bar{e}_1^0[\text{thisClass}])S \).

Let \( C = C' \cup \{ \text{cast}(I, e) \} \). The result follows by rules (**ct-ucast**) and (**cc-ucast**).

Case (**T-ncast**). Then \( e = (I)e_1, \Gamma \vdash t e_1 : I_1 \) and \( I_1 \not< : I \). By induction

- \( \langle \text{FD} : \sigma \rangle; \text{this}: \text{thisClass}, \bar{x} : \bar{e}_t e_1 : e_1 (\langle \text{FD} : \sigma' \rangle) : C, \vdash \sigma e_1 = [C/\text{thisClass}] : S \) and \( I_1 = (\bar{e}_1^0[\text{thisClass}])S \).

Let \( C = C' \cup \{ \text{cast}(I, e) \} \). The result follows by rules (**ct-ncast**) and (**cc-ncast**).

\[
\]
Lemma C.3. For every class definition \( \text{class } C \text{ implements } \bar{t} \) by \( \{ CD; \bullet, MD \} \{ FD; \} \) it holds that: \( \vdash_t CD \text{ OK} \) if and only if \( \vdash_{ct} CD \text{ OK} \).

PROOF. By Lemma C.2.

PROOF OF THEOREM 4.3 (Equivalence of \( \vdash_{ct}\)-typability and \( \vdash_t\)-typability on FFPTJ programs). By Lemmas C.3 and C.1.

References