On Traits and Types in a Java-like Setting

Viviana Bono, Ferruccio Damiani, and Elena Giachino

Dipartimento di Informatica, Università di Torino
{bono,damiani,giachino}@di.unito.it

Abstract. Both single and multiple class-based inheritance are often inappropriate as a reuse mechanism, because classes play two competing roles. Namely, a class is both a *generator of instances* and a *unit of reuse*. *Traits* are composable pure units of behavior reuse, consisting only of methods, that have been proposed as an add-on to single class-based inheritance in order to improve reuse. However, adopting traits as an add-on to traditional class-based inheritance is not enough: classes, besides their primary role of generators of instances, still play the competing role of units of reuse. Therefore, a style of programming oriented to reuse is not enforced by the language, but left to the programmer's skills. Traits have been originally proposed in the setting of dynamically typed language. When static typing is also taken into account, the role of unit of reuse and the role of type are competing, too.
We argue that, in order to support the development of reusable program components, object oriented programming languages should be designed according to the principle that *each software structuring construct must have exactly one role*. We propose a realignment of the class-based object-oriented paradigm by presenting programming language features that separate completely the declarations of object *type*, *behavior* and *generator*. We illustrate our proposal through a core calculus and prove the soundness of the type system w.r.t. the operational semantics.

Key words: Type System, Inheritance, Composition, Flattening.

1 Introduction

It is common opinion that standard class-based inheritance does not support low coupling and, therefore, does not support well code reuse. This phenomenon is often described as the *fragile base-class problem* and it is well-described in the work by Mikhajlov and Sekerinski [20]. A well-known technique to circumvent the fragile base-class problem is to promote the use of interface-based polymorphism. This idea is also present in most of the design patterns, such as the GoF design patterns [14], in order to make the patterns as higher-level as possible with respect to the implementation details.

Class-based inheritance was criticized again recently by Schärli et al. [25, 10], by pointing out that both single and multiple class-based inheritance are often inappropriate as a reuse mechanism. They identify the problem in the fact that classes play two competing roles. Namely, a class is both a *generator of instances* (hence it must provide a *complete* set of basic features) and a *unit of reuse*

Please use the following format when citing this chapter:

Bono, V., Damiani, F. and Giachino, E., 2008, in IFIP International Federation for Information Processing, Volume 273; Fifth IFIP International Conference on Theoretical Computer Science; Giorgio Ausiello, Juhani Karhumäki, Giancarlo Mauri, Luke Ong; (Boston: Springer), pp. 367–382.

(hence it should provide a *minimal* set of sensibly reusable features). Schärli et al. also observed that *mixins* [7, 17, 13, 3], which are subclasses parameterized over their superclasses, are not necessarily appropriate for composing units of reuse. The problem is due to the fact that, being based on the ordinary single inheritance operator, mixing composition is linear. Indeed, the formulation of mixins given by Bracha in JIGSAW [6] does not suffer of this problem,[1] but most of the subsequent formulations of the mixin construct do.

To overcome these problems, Schärli et al. proposed *traits*, composable pure units of behavior reuse consisting only of methods, that can be composed in an arbitrary order via operations ensuring that the composite unit (trait or class) has complete control over the composition and must resolve conflicts explicitly. However, both in the original proposal and (to the best of our knowledge) in all the trait-based approaches that can be found in the literature (with the exception of the FORTRESS language proposal [1], currently under development), traits live together with the traditional class-based inheritance. Therefore, besides their primary role of generators of instances, classes can still play the competing role of units of reuse, and a style of programming oriented to reuse is not enforced by the language, but left to the programmer's skills.

The original proposal of Schärli et al. does not address typing issues. Various proposals for using traits in connection with static typing can be found in the literature (we refer to [21] for a brief overview). In some of these proposals (notably in the SCALA [22] and in the FORTRESS [1] languages) each trait, like each class, also defines a type. However, as a matter of fact, the role of unit of reuse and the role of type are competing. For instance, in order be able to define the subtyping relation on traits in such a way that a trait (or a class) is always a subtype of the component traits, SCALA and FORTRESS rule out operations on traits such as method exclusion and renaming, limiting the reuse potential of traits. The distinction between the role of type and the role of unit of reuse, described in terms of type and class, dates back at least to Snyder [27] (see also Cook et al. [9]).

Having in mind the need of promoting interface-based polymorphism and arbitrarily composable units of reuse, we would like to go further and give classes the role of object generators only.

We argue that, in order to support the development of reusable program components, object oriented programming languages should be designed according to the principle that *each software structuring construct must have exactly one role*. We propose programming language features that separate completely the declarations of object *type*, *behavior* and *generator*. Namely, we consider:

- *Interfaces*, as pure types.
- *Traits*, as pure units of behavior reuse.
- *Classes*, as pure generators of instances.

[1] JIGSAW introduces very general operators for module manipulation. Some of them have been later, independently, developed for traits.

Interfaces can be defined by extending other interfaces (the interface hierarchy induces subtyping). Traits can be defined by composing other traits. Classes are defined by composing traits, implementing interfaces, and defining fields.

Note that there are no hierarchical dependencies among classes. Therefore, a first outcome of the complete role separation is that problems of fragility in a class hierarchy (that arise with class-based and mixin-based inheritance) are avoided *a priori*: there is no class hierarchy. Since traits and classes do not define types, another outcome of the complete role separation is that the use of operations like method exclusion and renaming is not limited by the need of ensuring that each trait (or class) is a subtype of the composing traits (see Sect. 2).

Recently, Bergel et al. [4] pointed out several limitations of the trait model. In order to overcome these limitations, they propose (in a SMALLTALK-like setting) to make traits *stateful* by allowing traits to have private fields that, through a variable access operator, may be accessed from the clients possibly under a new name, and possibly merged with other variables. Our proposal provides (in a JAVA-like setting) an alternative solution to the limitations of the stateless trait model. Also, Bergel et al. observed that: "An open question for further study is whether trait composition can subsume class-based inheritance, leading to a programming language based on composition rather than inheritance as the primary mechanism for structuring code following JIGSAW [6] design." Our investigation addresses the previous question by providing a foundation for a realignment of the class-based object-oriented paradigm to support the systematic structuring of code in "single-role" reusable units. Besides their power of reuse, traits have attracted a great deal of attention in the programming language research community because of their simple semantics. We believe that our proposal is a step forward towards simplicity.

A preliminary version of the results presented in this paper appeared as [5].

Organization of the Paper. Section 2 illustrates our proposal through an example. Section 3 presents the syntax of FRJ (a core calculus for reusable units based on the constructs introduced above), outlines its type system and its operational semantics, and states a type soundness result. We conclude by discussing some related work and outlining possible directions for further work.

2 An Example

In this section we provide a simple example of code that cannot lead to unanticipated reuse both in traditional class-based languages and in trait-based languages where a composite trait is a subtype of the component traits, but that can be reused in an unanticipated way in a language based on our proposal. We exploit a standard Java-like notation, in particular we use a more general syntax for constructors than the one that will be presented in Section 3.1.

Consider the task of developing a class `Stack` that implements the interface:

```
interface IStack { boolean isEmpty(); void push(Object o); Object pop(); }
```

In a traditional class-based language (like, e.g., JAVA) it is natural to write a class like:

```
class Stack implements IStack { List l; Stack() { l=new LinkedList(); }
   boolean isEmpty() { return (l.size() == 0); }
   void push(Object o) { l.addFirst(); }
   Object pop() { Object o=l.getFirst(); l.removeFirst(); return o; }
}
```

Suppose that later on it becomes necessary to develop a class `Stack'` that implements the interface:

```
interface IStack' { Boolean isEmpty(); void push(Object o); void pop();
                    Object top(); }
```

In a traditional class-based language there is no straightforward way to reuse the code in class `Stack` and the simplest thing to do is to write a class like:

```
class Stack' implements IStack' { List l; Stack'() { l=new LinkedList(); }
   boolean isEmpty() { return (l.size() == 0); }
   void push(Object o) { l.addFirst(); }
   void pop() { l.removeFirst(); }
   Object top() { return l.getFirst(); }
}
```

To illustrate our proposal, we exploit a JAVA-like syntax (we still do not have an implementation). In a class the only (implicitly) public methods are those declared in the interfaces implemented by the class. All the other methods and the fields are (implicitly) private. All the constructors must be declared and are (implicitly) public. Moreover, for every library class (such as `Object`, `Integer`, etc.) we assume an interface and a trait. The same name can be used to denote the interface, the trait and the class. The `Object` interface is implicity extended by any interface and the `Object` trait is implicity used by any class.

A class `Stack`, whose instance type is the interface `IStack`, can be naturally written by defining separately instance behaviour and generation as follows:

```
trait TStack is { List l;
   boolean isEmpty() { return (l.size() == 0); }
   void push(Object o) { l.addFirst(); }
   Object pop() { Object o=l.getFirst(); l.removeFirst(); return o; } }

class Stack implements IStack by TStack
   { List l; Stack() { l=new LinkedList(); } }
```

A class `Stack'` that implements the interface `IStack'` can be straightforwardly written as follows by defining a trait `TStack'` that reuses the trait `TStack`:

```
trait TStack' is (TStack exclude pop)
               + { List l;  void pop() { l.removeFirst(); }
                            Object top() { return l.getFirst(); } }

class Stack' implements IStack' by TStack'
   { List l; Stack'() { l=new LinkedList(); } }
```

```
ID ::= interface I extends Ī { S̄; }
S  ::= I m (Ī x̄)

TD  ::=  trait T is TE
TE  ::=  { F̄; S̄; M̄ } | T | TE + TE | TE exclude m | TE alias m as m
       | TE duplicate m as m | TE rename m to m | TE rename f to f
F  ::= I f
M  ::= S { return e; }
e  ::= x | e.f | e.m(ē) | new C(ē) |(I)e

CD ::= class C implements Ī by TE { F̄; K }
K  ::= C(Ī f̄) { this.f̄ = f̄; }
```

Fig. 1 FRJ: Syntax

The trait `TStack'` above can be alternatively defined as follows:

```
trait TStack' is (TStack rename pop to poptop)
               + { Object poptop(); void push(Object);
                   void pop() { poptop(); }
                   Object top() { Object o=poptop(); push(o); return o; } }
```

Note that, if traits were types and composed traits were subtypes of the component traits, both the declarations of the trait `TStack'` would not typecheck.

3 FRJ: a Calculus for Reusable Units

In this section we provide a formal account of our idea by presenting FRJ (Featherweight Reusable Java), a minimal core calculus for interfaces, traits and classes, in the spirit of FJ (Featherweight Java) [15].

3.1 Syntax

The syntax of our calculus, FRJ, is presented in Fig. 1. We also consider a calculus, FFRJ (Flat FRJ), obtained by removing the portions of the syntax highlighted in grey.

We use the overbar sequence notation according to [15]. For instance: "$\bar{\mathtt{f}}$" denotes the possibly empty sequence "$\mathtt{f}_1, ..., \mathtt{f}_n$", the pair "$\bar{\mathtt{I}}\,\bar{\mathtt{x}}$" stands for "$\mathtt{I}_1\,\mathtt{x}_1, ..., \mathtt{I}_n\,\mathtt{x}_n$", "$\bar{\mathtt{I}}\,\bar{\mathtt{f}}$;" stands for "$\mathtt{I}_1\,\mathtt{f}_1; ...; \mathtt{I}_n\,\mathtt{f}_n$;", and the assignment "$\mathtt{this}.\bar{\mathtt{f}} = \bar{\mathtt{f}}$;" stands for "$\mathtt{this}.\mathtt{f}_1 = \mathtt{f}_1; ...; \mathtt{this}.\mathtt{f}_n = \mathtt{f}_n$;". The empty sequence is denoted by "$\bullet$".

Sequences of named elements (e.g., methods signatures, fields declarations,...) are assumed to contain no duplicate names, the sequence of the names of the elements of $\overline{\mathtt{S}}$ is denoted by $\mathit{names}(\overline{\mathtt{S}})$, the subsequence of the elements of $\overline{\mathtt{S}}$ with

the names $\bar{\mathtt{n}}$ is denoted by $\mathit{extract}(\bar{\mathtt{n}}, \overline{\mathtt{S}})$, and $\mathit{discard}(\bar{\mathtt{n}}, \overline{\mathtt{S}})$ denotes the sequence obtained from $\overline{\mathtt{S}}$ by removing the elements with the names $\bar{\mathtt{n}}$. Following [15], we use a set-based notation for operators over sequences of named elements. For instance, $\mathtt{M} = \mathtt{I\,m}\,(\bar{\mathtt{I}}\,\mathtt{x})\{\mathtt{return\ e}\} \in \bar{\mathtt{M}}$ means that the method declaration $\mathtt{M}$ occurs in $\bar{\mathtt{M}}$. In the union and in the intersection of sequences of named elements, denoted by $\overline{\mathtt{S}} \cup \overline{\mathtt{Z}}$ and $\overline{\mathtt{S}} \cap \overline{\mathtt{Z}}$, respectively, it is assumed that if $\mathtt{n} \in \mathit{names}(\overline{\mathtt{S}})$ and $\mathtt{n} \in \mathit{names}(\overline{\mathtt{Z}})$ then $\mathit{extract}(\mathtt{n}, \overline{\mathtt{S}}) = \mathit{extract}(\mathtt{n}, \overline{\mathtt{Z}})$.

The concatenation of two sequences $\overline{\mathtt{S}}$ and $\overline{\mathtt{Z}}$ is denoted by $\overline{\mathtt{S}} \cdot \overline{\mathtt{Z}}$, where, if $\overline{\mathtt{S}}$ and $\overline{\mathtt{Z}}$ are sequences of named elements, it is assumed that $\mathit{names}(\overline{\mathtt{S}}) \cap \mathit{names}(\overline{\mathtt{Z}}) = \emptyset$.

A class table $\mathtt{CT}$ is a map from class names to class declarations. Similarly, an interface table $\mathtt{IT}$ and a trait table $\mathtt{TT}$ map interface and trait names to interface and trait declarations, respectively. A FRJ program is a 4-tuple $(\mathtt{IT}, \mathtt{TT}, \mathtt{CT}, \mathtt{e})$. In presenting the type system and the flattening translation we assume fixed, global tables $\mathtt{IT}$, $\mathtt{TT}$, and $\mathtt{CT}$. We also assume that these tables are *well-formed*, i.e., they contain an entry for each interface/trait/class mentioned in the program, and the interface subtyping and trait reuse graphs are acyclic.

The distinguishing features of FRJ w.r.t. the original trait proposal [10] and to other proposals of traits for JAVA-like setting [26, 18, 21] are the following:

- Classes and traits are not types and class-based inheritance is not present.
- Traits (and classes) can be typechecked in isolation (as in Chai_2 [26]).
- A basic trait expression $\{\,\bar{\mathtt{F}};\ \overline{\mathtt{S}};\ \bar{\mathtt{M}}\,\}$ provides the methods $\bar{\mathtt{M}}$ and declares the type of the required fields $\bar{\mathtt{F}}$ and methods $\overline{\mathtt{S}}$ (that can can be directly accessed by the bodies of the methods $\bar{\mathtt{M}}$).[2]
- In the *symmetric sum* operation (that merges two traits to form a new trait) we require that the summed traits must be disjoint (that is, they must not provide identically named methods).[3]
- The operation *exclude*, that forms a new trait by removing a method from an existing trait, is the usual one (i.e., as in [10, 26, 18, 21]).
- We have the operations *alias* and *duplicate* that form a new trait by giving a new name to an existing method. The two operations are identical on non-recursive methods. When a recursive method is aliased, its recursive invocation refers to the original method (as in [10]). When a recursive method is duplicated, its recursive invocation refers to the duplicate (as in the interpretation of aliasing proposed in [18]).
- We have the operation *rename* that 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.[4]

[2] Field requirements were not present in [10] and in [26, 18, 21]. They have been introduced in [12] in the setting of ML-like languages.

[3] According to other proposals, two methods with the same name do not conflict if they are syntactically equal [10, 21] or if they originate from the same subtrait [18].

[4] Method renaming is not present in [10] and in [26, 18, 21]. It has been introduced in [23] in the setting of ML-like languages. At the best of our knowledge, required field renaming is new.

– The *override* operation, that layers additional methods over an existing trait, is not present. It can be simulated by exclusion and symmetric sum.
– We use *interfaces* to explicitly declare the public methods of a class.

3.2 Typing

The FRJ type system combines nominal and structural typing. Within a basic trait expression, the uses of method parameters are type-checked according to the nominal notion of typing defined by the interface hierarchy, while the uses of the `this` metavariable are type-checked according to a structural notion of typing that takes into account the field and methods *required* by the trait and the methods *provided* by the trait.

3.2.1 Types, Constraints and Subtying. *Pure signatures*, ranged over by σ and ζ, are method signatures deprived of parameter names. For instance, the pure signature associated to the signature $\mathtt{I\,m}(\mathtt{I}_1\,\mathtt{x}_1, ..., \mathtt{I}_n\,\mathtt{x}_n)$ is $\mathtt{I\,m}(\mathtt{I}_1, ..., \mathtt{I}_n)$.

The syntax of *nominal types* is as follows: $\boxed{\eta \ ::= \ \mathtt{C} \mid \mathtt{I}}$. (I.e., a nominal type is either a class name or an interface name.) The syntax of *types for expressions* is as follows: $\boxed{\theta \ ::= \ \langle\, \bar{\mathtt{F}} \ \vdash \ \overline{\sigma} \,\rangle \mid \eta}$. The type of the expression `this` is a pair $\langle\, \bar{\mathtt{F}} \ \vdash \ \overline{\sigma} \,\rangle$, specifying that `this` has the fields $\bar{\mathtt{F}}$ and methods with (pure) signatures $\overline{\sigma}$. The type of an object creation expression $\mathtt{new\,C}(\cdots)$ is the class `C`. The type of any other expressions `e` is an interface name.

Besides assigning to each expression `e` a type describing the object yielded by the evaluation of `e`, the FRJ type system infers also the constraints on `this` imposed by its use within `e`. *Constraints*, ranged over by γ, are triples $\langle\, \bar{\mathtt{F}} \ \vdash \ \overline{\sigma} \ \vdash \ \bar{\mathtt{I}} \,\rangle$ specifying that the expression `e` selects the fields $\bar{\mathtt{F}}$ and the methods $\overline{\sigma}$ on `this`, and requires that `this` has the nominal types (interfaces) $\bar{\mathtt{I}}$. In particular, the interfaces in $\bar{\mathtt{I}}$ are the types of the method formal parameters to which `this` is passed inside the expression `e`. We recall that `this` will assume a meaning according to the class where the traits will be used. The typing rule for classes will check that such a class satisfies the constraints inferred for the bodies of the methods declared in the composing traits.

The *subtyping relation* for nominal types is the reflexive and transitive closure of the interface implementatation/extension relation declared by the `implements` clauses in the class table `CT` and by the `extends` clauses in the interface table `IT`. It is formalized by the judgement $\boxed{\eta_1 <: \eta_2}$ to be read: "η_1 is a subtype of η_2".

3.2.2 Typing Rules. An environment Γ is either a finite mapping form variable names (including `this`) to types, written "$\bar{\mathtt{x}} : \bar{\mathtt{I}}$, $\mathtt{this} : \langle\, \bar{\mathtt{F}} \ \vdash \ \overline{\sigma} \,\rangle$", or the empty mapping, written "$\bullet$". The typing rules for interface declarations, expressions, method declarations, trait declarations and class declarations are syntax directed, with one rule for each form of term, except that (following [15]) there

are three different rules for casts (to distinguish between *upcasts*, *downcasts*, and *stupid casts*). The typing judgements are the following:

- $\boxed{\vdash \texttt{interface I extends } \bar{\texttt{I}}\ \{\ \overline{\texttt{S}};\ \}\quad \text{OK}}$ to be read: "the declaration of the interface `I` is well-typed".
- $\boxed{\Gamma \vdash \texttt{e} : \theta \ \triangleright\ \gamma}$ to be read: "under the assumption in Γ, the expression `e` is well-typed with type θ and constraints γ".
- $\boxed{\texttt{this} : \langle\ \bar{\texttt{F}}\ \triangleright\ \overline{\sigma}\ \rangle \vdash \texttt{I m } (\bar{\texttt{I}}\ \bar{\texttt{x}})\{\texttt{return e;}\} : \mu}$ where $\mu = \zeta\ \triangleright\ \gamma$. To be read: "under the assumption that `this` has fields $\bar{\texttt{F}}$ and methods $\overline{\sigma}$, the declaration of method `m` is well-typed with type μ". I.e., the method `m` has signature ζ and its body enforces the constraints γ.
- $\boxed{\vdash \texttt{TE} : \bar{\mu}}$ where $\bar{\mu} = \mu_1 ... \mu_n \quad (n \geq 0)$. To be read: "the trait expression `TE` is well-typed with type $\bar{\mu}$". I.e., `TE` provides n methods with types $\mu_1, ..., \mu_n$, respectively.
- $\boxed{\vdash \texttt{trait T is TE} : \bar{\mu}}$ to be read: "the declaration of trait `T` is well-typed with type $\bar{\mu}$".
- $\boxed{\vdash \texttt{class C implements } \bar{\texttt{I}} \texttt{ by TE } \{\ \bar{\texttt{F}};\ \texttt{K}\ \}\quad \text{OK}}$ to be read: "the declaration of the class `C` is well-typed".

Note that, within a basic trait expression $\{\ \bar{\texttt{F}};\ \overline{\texttt{S}};\ \bar{\texttt{M}}\ \}$, we ask the programmer to declare exactly the fields $\bar{\texttt{F}}$ and the methods $\overline{\texttt{S}}$ selected on `this` within the method bodies in $\bar{\texttt{M}}$. Declaring the types of fields and methods has two benefits: (*i*) it provides a form of documentation that enforces awareness of what it is actually used in a program; (*ii*) it simplifies the inferred constraints. We decided not to ask to declare the name of the interfaces that are used as types of `this` within the method bodies in $\bar{\texttt{M}}$, as this would not introduce any benefits to counterbalance the overhead.

3.2.3 Well-typed FRJ ***programs.*** We write $\vdash_{\text{FRJ}} (\texttt{IT}, \texttt{TT}, \texttt{CT}, \texttt{e}) : \eta$, to be read: "the program $(\texttt{IT}, \texttt{TT}, \texttt{CT}, \texttt{e})$ is well-typed with type η", to mean that the interfaces in `IT`, the traits in `TT` and the classes in `CT` are well-typed, and the expression `e` is well typed with type η and empty constraints under the empty set of assumptions (i.e., the judgement $\bullet \vdash \texttt{e} : \eta\ \triangleright\ \langle\ \bullet\ \triangleright\ \bullet\ \triangleright\ \bullet\ \rangle$ holds).

3.3 Flattening and Reduction

Our traits enjoy the *flattening property* [21], i.e., when a class uses a trait the semantics of the methods defined within the trait declaration is the same as if the methods were defined within the class declaration.[5] The semantics of FRJ is specified by means of a flattening translation that maps a FRJ program into a FFRJ program and of a reduction semantics for FFRJ programs.

[5] Flattening just aims to provide a canonical semantics to traits, it is not an especially effective implementation technique.

$$[\![\texttt{class C implements } \bar{\texttt{I}} \texttt{ by TE } \{ \bar{\texttt{F}};\ \texttt{K} \}]\!] \stackrel{\text{def}}{=} \texttt{class C implements } \bar{\texttt{I}} \texttt{ by } \{ \bar{\texttt{F}};\ \bullet;\ [\![\texttt{TE}]\!] \} \{ \bar{\texttt{F}};\ \texttt{K} \}$$

$$\begin{aligned}
[\![\{ \bar{\texttt{F}};\ \bar{\texttt{S}};\ \bar{\texttt{M}} \}]\!] &\stackrel{\text{def}}{=} \bar{\texttt{M}} \\
[\![\texttt{T}]\!] &\stackrel{\text{def}}{=} [\![\texttt{TE}]\!] \qquad \textbf{if } \texttt{TT(T)} = \texttt{trait T is TE} \\
[\![\texttt{TE}_1 + \texttt{TE}_2]\!] &\stackrel{\text{def}}{=} [\![\texttt{TE}_1]\!] \cdot [\![\texttt{TE}_2]\!] \\
[\![\texttt{TE exclude m}]\!] &\stackrel{\text{def}}{=} \mathit{discard}(\texttt{m}, [\![\texttt{TE}]\!]) \\
[\![\texttt{TE alias m as m}']\!] &\stackrel{\text{def}}{=} \bar{\texttt{M}} \cdot (\texttt{I m}'(\bar{\texttt{I}}\ \bar{\texttt{x}})\{\texttt{return e;}\}) \\
&\qquad \textbf{if } [\![\texttt{TE}]\!] = \bar{\texttt{M}} \textbf{ and } \texttt{I m}(\bar{\texttt{I}}\ \bar{\texttt{x}})\{\texttt{return e;}\} \in \bar{\texttt{M}} \\
[\![\texttt{TE duplicate m as m}']\!] &\stackrel{\text{def}}{=} \bar{\texttt{M}} \cdot (\texttt{I m}'(\bar{\texttt{I}}\ \bar{\texttt{x}})\{\texttt{return e}[^{\texttt{this.m}'}/_{\texttt{this.m}}]\texttt{;}\}) \\
&\qquad \textbf{if } [\![\texttt{TE}]\!] = \bar{\texttt{M}} \textbf{ and } \texttt{I m}(\bar{\texttt{I}}\ \bar{\texttt{x}})\{\texttt{return e;}\} \in \bar{\texttt{M}} \\
[\![\texttt{TE rename f to f}']\!] &\stackrel{\text{def}}{=} [\![\texttt{TE}]\!][\texttt{f}'/\texttt{f}] \\
[\![\texttt{TE rename m to m}']\!] &\stackrel{\text{def}}{=} \mathit{mR}([\![\texttt{TE}]\!], \texttt{m}, \texttt{m}') \\
\\
\mathit{mR}(\texttt{I n}(\bar{\texttt{I}}\ \bar{\texttt{x}})\{\texttt{return e;}\}, \texttt{m}, \texttt{m}') &\stackrel{\text{def}}{=} \texttt{I n}[\texttt{m}'/\texttt{m}](\bar{\texttt{I}}\ \bar{\texttt{x}})\{\texttt{return e}[^{\texttt{this.m}'}/_{\texttt{this.m}}]\texttt{;}\} \\
\mathit{mR}(\texttt{M}_1 \cdot \ldots \cdot \texttt{M}_n, \texttt{m}, \texttt{m}') &\stackrel{\text{def}}{=} (\mathit{mR}(\texttt{M}_1, \texttt{m}, \texttt{m}')) \cdot \ldots \cdot (\mathit{mR}(\texttt{M}_n, \texttt{m}, \texttt{m}'))
\end{aligned}$$

Fig. 2 Flattening FRJ to FFRJ

3.3.1 Flattening Translation for FRJ**.** A FFRJ program is a FRJ program with an empty trait table. The translation removes the trait table and replaces the class table with a suitable one containing only FFRJ classes. The translation is specified through the function $[\![\cdot]\!]$, given in Fig. 2, that maps a FRJ class declaration to a FFRJ class declaration and a trait expression to a sequence of method declarations. We will write $[\![\texttt{CT}]\!]$ to denote the class table containing the translation of all the classes in CT. The clauses in Fig. 2 are self-explanatory. Note that the clause for 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.

3.3.2 Reduction for FFRJ**.** A FFRJ program is a 4-tuple $(\texttt{IT}, \bullet, \texttt{CT}, \texttt{e})$. A FFRJ class "class C implements $\bar{\texttt{I}}$ by $\{ \bar{\texttt{F}};\ \bullet;\ \bar{\texttt{M}} \}$ $\{ \bar{\texttt{F}};\ \texttt{K} \}$" can be understood as the JAVA class "class C implements $\bar{\texttt{I}}$ $\{ \bar{\texttt{F}};\ \texttt{K}\ \bar{\texttt{M}} \}$". Following FJ [15], we give the semantics of FFRJ by means of a reduction relation of the form $\texttt{e} \rightarrow \texttt{e}'$, to be read "expression e reduces to expression e′ in one step". We write $\rightarrow^\star$ to denote the reflexive and transitive of $\rightarrow$. Values are defined by the following syntax: $\boxed{\texttt{v} \ ::= \ \texttt{new C}(\bar{\texttt{v}})}$.

3.4 Properties

The flattening translation preserves the type of programs.

Theorem 1 (Flattening Preserves the Type of Programs).
If $\vdash_{\text{FRJ}} (\texttt{IT}, \texttt{TT}, \texttt{CT}, \texttt{e}) : \eta$*, then* $\vdash_{\text{FRJ}} (\texttt{IT}, \bullet, [\![\texttt{CT}]\!], \texttt{e}) : \eta$.

To prove the type soundness result for FFRJ we need to consider a suitable notion of typing for runtime expressions. As for FJ [15], the syntax of runtime expressions is the same of expressions. Constraints are not needed to prove the type soundness for FFRJ, therefore the typing for runtime expressions do not consider constraints. An environment for runtime expressions Δ is either a finite mapping from variable names (including `this`) to types, written "$\overline{\mathtt{x}} : \overline{\mathtt{I}}$, `this` : `C`", or the empty mapping, written "$\bullet$". The typing judgement for runtime expressions is $\boxed{\Delta \vdash' \mathtt{e} : \eta}$ to be read: "under the assumption in Δ, the runtime expression `e` is well-typed with type η".

The type soundness result comes in two parts: first it relates the typing of expressions with the typing of runtime expressions, then it proves the type soundness with respect to the runtime expression typing.

Theorem 2 (Well-typed FFRJ **expressions are well-typed runtime expression with a more specific type).**
If $\bullet \vdash \mathtt{e} : \eta$*, then* $\bullet \vdash' \mathtt{e} : \eta'$ *for some* η' *such that* $\eta' <: \eta$*.*

The following theorem can be proved by using the standard technique of subject reduction and soundness theorems.

Theorem 3 (FFRJ **Type Soundness).**
If $\bullet \vdash' \mathtt{e} : \eta$ *and* $\mathtt{e} \rightarrow^{\star} \mathtt{e}'$ *with* $\mathtt{e}'$ *a normal form, then* $\mathtt{e}'$ *is: Either a value* `v` *with* $\bullet \vdash' \mathtt{v} : \mathtt{C}$ *and* $\mathtt{C} <: \eta$*; Or an expression containing* $(\mathtt{I})\mathtt{new}\,\mathtt{C}(\overline{\mathtt{e}})$ *where* $\mathtt{C} \not<: \mathtt{I}$*.*

Following FJ [15], we say that a well-typed program $(\mathtt{IT}, \mathtt{TT}, \mathtt{CT}, \mathtt{e})$ is *cast-safe* if the type derivations involved in $\vdash_{\mathrm{FRJ}} (\mathtt{IT}, \mathtt{TT}, \mathtt{CT}, \mathtt{e}) : \eta$ include no downcasts or stupid casts. The following results hold.

Theorem 4 (Flattening Preserves Cast-Safeness).
If $(\mathtt{IT}, \mathtt{TT}, \mathtt{CT}, \mathtt{e})$ *is cast-safe, then* $(\mathtt{IT}, \bullet, [\![\mathtt{CT}]\!], \mathtt{e})$ *is cast-safe.*

Theorem 5 (No Typecast Errors in Cast-Safe FFRJ **Programs).**
If $(\mathtt{IT}, \bullet, [\![\mathtt{CT}]\!], \mathtt{e})$ *is cast-safe and* $\mathtt{e} \rightarrow^{\star} \mathtt{e}'$ *with* $\mathtt{e}'$ *a normal form, then* $\mathtt{e}'$ *is a value.*

4 Conclusions, Related and Further Work

The competing roles played by the same software structuring construct complicate the semantics and limits the reuse potential in mainstream object-oriented class-based programming languages.

To the best of our knowledge, the conflict between the roles of *unit of reuse* and *generator of instances* was firstly described by Schärli et al. [25, 10]. We claim also that the roles of *unit of reuse* and *type* are competing (see Sect. 1). In this respect, we propose to increase both the simplicity and the flexibility of the object-oriented paradigm by adopting programming language features that

separate completely the declarations of object *type*, *behavior*, and *generator*. We developed a hybrid nominal/structural type system that allows to typecheck traits in isolation and proved its soundness.

The literature related to our proposal has been partially quoted through the paper. We add here comparisons and remarks concerning the type system and the recent proposals on *trait-based metaproprograming* [24] and *stateful traits* [4].

Sophisticated hybrid nominal/structural type systems have been already proposed [11, 19, 24]. In particular, in [24], the combination of nominal and structural types is conceptually similar to ours, but exploited at a different level. Namely, it is exploited to type trait functions, that provide a mechanism (termed trait-based metaproprograming) to obtain reusable class-member-level patterns. Another important difference between our proposal and the one in [24] is that, in the latter, traits play also the competing role of type, which instead we want to avoid.

Stateful traits [4] were introduced (in the setting of SMALLTALK-like languages) to avoid duplication of code connected directly with field initialization and manipulation. Our traits are stateless, however, since they can have required fields, it is possible to avoid the same kind of duplication of code that motivated the introduction of stateful traits. Moreover required fields names are unimportant because we provide a field rename operation. As byproducts, since required field renaming works synergically with method renaming, exclusion, aliasing, and duplication, we obtain more reuse potential.

In further work, we would like to formulate our proposal in a SMALLTALK-like setting (this would allow a careful comparison with the stateful trait proposal), to extend our type system to deal with generics, and to adapt our proposal to deal with *dynamic trait substitution* (see *Chai*$_3$ [26]). We also plan to develop prototypical implementations.

A special form of *reuse* is at the base of the contemporary *agile* software development methodologies [2]. Such methodologies are based on an iterative approach, where each iteration may include all of the phases necessary to release a small increment of a new functionality: planning, requirements analysis, design, coding, testing, and documentation. While an iteration may not add enough functionality to guarantee the release of a final product, an agile software project intends to be capable of releasing new software at the end of every iteration, but this means that the next iteration will *reuse* the software produced in the previous ones. We believe that an interesting future research direction is to investigate whether the programming language features proposed in this paper may help in writing software following an agile methodology. In this respect, we plan both to develop a trait-oriented agile methodology, suitable to be used directly within trait-based languages, and some trait-mining strategies, in order to re-engineer class-based libraries into trait-based ones. Some work in this respect was already done in the SMALLTALK-like setting [8, 16].

4.0.1 Acknowledgements. We thank Davide Ancona, Stéphane Ducasse, Paola Giannini, Oscar Nierstrasz, Betti Vennneri, Elena Zucca and the anonymous referees for comments, suggestions, and bibliographic references.

References

1. The Fortress language specification. http://research.sun.com/projects/plrg/fortress.pdf.
2. The Agile Alliance. Manifesto for Agile Software Development. `http://agilemanifesto.org/`.
3. D. Ancona, G. Lagorio, and E. Zucca. Jam—designing a Java extension with mixins. *ACM TOPLAS*, 25(5):641–712, September 2003.
4. A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits. In *Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006)*, volume 4406 of *LNCS*, pages 66–90. Springer, 2007.
5. V. Bono, F. Damiani, and E. Giachino. Separating Type, Behavior, and State to Achieve Very Fine-grained Reuse. In *Electronic proceedings of FTfJP'07 (http://www.cs.ru.nl/ftfjp/)*, 2007.
6. G. Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.
7. G. Bracha and W. Cook. Mixin-based inheritance. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1990*, volume 25(10) of *SIGPLAN Notices*, pages 303–311. ACM Press, October 1990.
8. D. Cassou, S. Ducasse, and R. Wuyts. Redesigning with traits: the nile stream trait-based library. In *Proc. International Conference on Dynamic Languages*, pages 50–79, 2007.
9. W.R. Cook, W.L. Hill, and P.S. Canning. Inheritance is not subtyping. In *ACM Symp. on Principles of Programming Languages 1990*, pages 125–135. ACM Press, 1990.
10. S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2):331–388, 2006.
11. K. Fisher and J. Reppy. Inheritance-based subtyping. *Information and Computation*, 177(1):28–55, 2002.
12. K. Fisher and J. Reppy. A typed calculus of traits. In *Electronic proceedings of FOOL 2004*, 2004.
13. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *ACM Symp. on Principles of Programming Languages 1998*, pages 171–183. ACM Press, 1998.
14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
15. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
16. A. Lienhard, S. Ducasse, and G. Arévalo. Identifying traits with formal concept analysis. In *Proc. 20th Conference on Automated Software Engineering (ASE'05)*, pages 66–75. IEEE Computer Society, 2005.
17. M. Van Limberghen and T. Mens. Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.
18. L. Liquori and A Spiwack. Feathertrait: A modest extension of featherweight java. *ACM TOPLAS*. To appear.
19. D. Maleyery and J. Aldrich. Combining structural subtyping and external dispatch. In *Electronic proceedings of FOOL/WOOD 2007*, 2007.
20. L. Mikhajlov and E. Sekerinski. A Study of the Fragile Base Class Problem. In *Proc. ECOOP '98*, volume 1445 of *LNCS*, pages 355–382. Springer-Verlag, 1998.

21. O. Nierstrasz, S. Ducasse, and N. Schärli. Flattening traits. *JOT (www.jot.fm)*, 5(4):129–148, 2006.
22. M. Odersky. The Scala Language Specification, version 2.4. Technical report, Programming Methods Laboratory, EPFL, Switzerland, 2007.
23. J. Reppy and A. Turon. A foundation for trait-based metaprogramming. In *Electronic proceedings of FOOL/WOOD 2006*, 2006.
24. J. Reppy and A. Turon. Metaprogramming with traits. In *ECOOP 2007*, volume 4609 of *LNCS*, pages 373–398. Springer, 2007.
25. N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *ECOOP 2003*, volume 2743 of *LNCS*, pages 248–274. Springer, 2003.
26. C. Smith and S. Drossopoulou. *Chai*: Traits for java-like languages. In *ECOOP'05*, LNCS 3586, pages 453–478. Springer, 2005.
27. A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1986*, volume 21(11) of *SIGPLAN Notices*, pages 38–45. ACM Press, 1986.

5 FRJ Typing Rules

The typing rules use the auxiliary functions given in Fig. 3: *fields* (that returns the sequence of the fields declared in a class C), *interfaces* (that when applied to an interface name I returns the name I itself, and when applied to a class name C returns the sequence of the interface names implemented by the class C), *methods* (that returns the sequence of the methods declared in a class C) and *mPSig* (that returns the sequence of the pure signatures of the methods associated to a sequence of method non-pure signatures, or interfaces, or method declarations).

Fields lookup (function *fields*)

$$fields(\mathtt{C}) = \bar{\mathtt{F}} \qquad \textbf{if } \mathtt{CT(C)} = \mathtt{class\ C}\ \cdots\ \{\ \bar{\mathtt{F}};\ \mathtt{C}(\bar{\mathtt{F}})\{\cdots\}\ \}$$

Interfaces lookup (function *interfaces*)

$$\begin{array}{ll} interfaces(\mathtt{C}) = \bar{\mathtt{I}} & \textbf{if } \mathtt{CT(C)} = \mathtt{class\ C\ implements}\ \bar{\mathtt{I}}\ \mathtt{by}\ \cdots \\ interfaces(\mathtt{I}) = \mathtt{I} & \end{array}$$

Methods lookup (function *methods*)

$$methods(\mathtt{C}) = \bar{\mathtt{M}} \qquad \textbf{if } \mathtt{CT(C)} = \mathtt{class\ C}\ \cdots\ \mathtt{by}\ \ \{\cdots;\ \bullet;\ \bar{\mathtt{M}}\}\ \{\cdots\}$$

Method pure signatures lookup (function *mPSig*)

$$\begin{array}{l} mPSig(\mathtt{I\ m}\ (\bar{\mathtt{I}}\ \bar{\mathtt{x}})) = \mathtt{I\ m}\ (\bar{\mathtt{I}}) \\ mPSig(\mathtt{S}_1;...;\mathtt{S}_n;) = mPSig(\mathtt{S}_1) \cdot ... \cdot mPSig(\mathtt{S}_n) \\ mPSig(\mathtt{I}) = mPSig(\bar{\mathtt{I}}) \cup mPSig(\overline{\mathtt{S}};) \qquad \textbf{if } \mathtt{IT(I)} = \mathtt{interface\ I\ extends}\ \bar{\mathtt{I}}\ \{\ \overline{\mathtt{S}};\ \} \\ mPSig(\mathtt{I}_1,...,\mathtt{I}_n) = mPSig(\mathtt{I}_1) \cup ... \cup mPSig(\mathtt{I}_n) \\ mPSig(\mathtt{S}\ \{\mathtt{return\ e};\}) = mPSig(\mathtt{S}) \\ mPSig(\mathtt{M}_1...\mathtt{M}_n) = mPSig(\mathtt{M}_1) \cdot ... \cdot mPSig(\mathtt{M}_n) \\ mPSig(\mathtt{C}) = mPSig(methods(\mathtt{C})) \end{array}$$

Fig. 3 FRJ: Auxiliary function *fields*, *interfaces*, *methods* and *mPSig*

The typing rule for interface declarations, the typing rules for expressions and method declarations, the typing rules for trait expressions and trait declarations, and the typing rule for class declarations are given in Fig.s 6, 4, 5 and 7. Some of the typing rules use assumptions of the form "E **ok**" to mean that the expression E (involving operations on sequences of named elements) yields a (well defined) sequence of named elements. For instance, the assertion "$(\overline{\sigma} \cup \overline{\zeta})$ **ok**" holds if and only if $\mathtt{n} \in \mathit{names}(\overline{\sigma})$ and $\mathtt{n} \in \mathit{names}(\overline{\zeta})$ imply $\mathit{extract}(\mathtt{n}, \overline{\sigma}) = \mathit{extract}(\mathtt{n}, \overline{\zeta})$.

Expression typing:

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

$$\frac{\Gamma \vdash \mathtt{this} : \langle\ \bar{\mathtt{F}}\ \vert\ \ldots\ \rangle \ \vert\ \langle \bullet \vert \bullet \vert \bullet \rangle \qquad \mathit{extract}(\mathtt{f}, \bar{\mathtt{F}}) = \mathtt{I\,f}}{\Gamma \vdash \mathtt{this.f} : \mathtt{I} \ \vert\ \langle\ \mathtt{I\,f}\ \vert\ \bullet\ \vert\ \bullet\ \rangle} \qquad \text{(T-Field)}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{e} : \theta \ \vert\ \langle\ \bar{\mathtt{F}}^{(0)}\ \vert\ \overline{\sigma}^{(0)}\ \vert\ \bar{\mathtt{I}}^{(0)}\ \rangle \\ \theta = \Gamma(\mathtt{this}) = \langle\ \ldots\ \vert\ \overline{\sigma}\ \rangle \ \textbf{implies}\ \mathtt{I\,m}\,(\mathtt{I}_1, \ldots, \mathtt{I}_n) = \mathit{extract}(\mathtt{m}, \overline{\sigma}) \\ \theta \neq \Gamma(\mathtt{this}) \ \textbf{implies}\ \mathtt{I\,m}\,(\mathtt{I}_1, \ldots, \mathtt{I}_n) = \mathit{extract}(\mathtt{m}, \mathit{mPSig}(\mathit{interfaces}(\theta))) \\ \forall i \in 1..n, \qquad \Gamma \vdash \mathtt{e}_i : \theta_i \ \vert\ \langle\ \bar{\mathtt{F}}^{(i)}\ \vert\ \overline{\sigma}^{(i)}\ \vert\ \bar{\mathtt{I}}^{(i)}\ \rangle \\ \mathcal{T} = \{i \mid i \in 1..n \ \textbf{and}\ \theta_i = \Gamma(\mathtt{this})\} \\ \forall i \in 1..n - \mathcal{T}, \quad \theta_i <: \mathtt{I}_i \qquad \forall i \in \mathcal{T}, \quad \overline{\sigma} \cup \mathit{mPSig}(\mathtt{I}_i)\ \textbf{ok}\end{array}}{\Gamma \vdash \mathtt{e.m}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \mathtt{I} \ \vert\ \langle\ \cup_{i \in 0..n} \bar{\mathtt{F}}^{(i)}\ \vert\ \cup_{i \in 0..n} \overline{\sigma}^{(i)}\ \vert\ (\cup_{i \in 0..n} \bar{\mathtt{I}}^{(i)}) \cup (\cup_{i \in \mathcal{T}} \mathtt{I}_i\ \rangle)} \qquad \text{(T-Invk)}$$

$$\frac{\begin{array}{l}\mathit{fields}(\mathtt{C}) = \mathtt{I}_1\,\mathtt{f}_1; \ldots; \mathtt{I}_n\,\mathtt{f}_n; \qquad \forall i \in 1..n, \quad \Gamma \vdash \mathtt{e}_i : \theta_i \ \vert\ \langle\ \bar{\mathtt{F}}^{(i)}\ \vert\ \overline{\sigma}^{(i)}\ \vert\ \bar{\mathtt{I}}^{(i)}\ \rangle \\ \mathcal{T} = \{i \mid i \in 1..n \ \textbf{and}\ \theta_i = \Gamma(\mathtt{this}) = \langle\ \ldots\ \vert\ \overline{\sigma}\ \rangle\} \\ \forall i \in 1..n - \mathcal{T}, \quad \theta_i <: \mathtt{I}_i \qquad \forall i \in \mathcal{T}, \quad \overline{\sigma} \cup \mathit{mPSig}(\mathtt{I}_i)\ \textbf{ok}\end{array}}{\Gamma \vdash \mathtt{new\,C}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \mathtt{C} \ \vert\ \langle\ \cup_{i \in 1..n} \bar{\mathtt{F}}^{(i)}\ \vert\ \cup_{i \in 1..n} \overline{\sigma}^{(i)}\ \vert\ (\cup_{i \in 1..n} \bar{\mathtt{I}}^{(i)}) \cup (\cup_{i \in \mathcal{T}} \mathtt{I}_i\ \rangle)} \qquad \text{(T-New)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \eta \ \vert\ \gamma \qquad \eta <: \mathtt{I}}{\Gamma \vdash (\mathtt{I})\mathtt{e} : \mathtt{I} \ \vert\ \gamma} \qquad \text{(T-UCast)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{J} \ \vert\ \gamma \qquad \mathtt{I} <: \mathtt{J} \qquad \mathtt{I} \neq \mathtt{J}}{\Gamma \vdash (\mathtt{I})\mathtt{e} : \mathtt{I} \ \vert\ \gamma} \qquad \text{(T-DCast)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \eta \ \vert\ \gamma \qquad \eta \not<: \mathtt{I} \qquad \mathtt{I} \not<: \eta \qquad \textbf{stupid warning}}{\Gamma \vdash (\mathtt{I})\mathtt{e} : \mathtt{I} \ \vert\ \gamma} \qquad \text{(T-SCast)}$$

Method declaration typing:

$$\frac{\begin{array}{l}\mathtt{this} : \langle\ \bar{\mathtt{F}}\ \vert\ \overline{\sigma}\ \rangle, \bar{\mathtt{x}} : \bar{\mathtt{J}} \vdash \mathtt{e} : \theta \ \vert\ \langle\ \bar{\mathtt{F}}'\ \vert\ \overline{\sigma}'\ \vert\ \bar{\mathtt{I}}\ \rangle \\ \theta = \langle\ \bar{\mathtt{F}}\ \vert\ \overline{\sigma}\ \rangle \ \textbf{implies}\ (\ \overline{\sigma} \cup \mathit{mPSig}(\mathtt{J})\ \textbf{ok and}\ \bar{\mathtt{I}}' = \bar{\mathtt{I}} \cup \mathtt{J}\ ) \\ \theta \neq \langle\ \bar{\mathtt{F}}\ \vert\ \overline{\sigma}\ \rangle \ \textbf{implies}\ (\ \theta <: \mathtt{J}\ \textbf{and}\ \bar{\mathtt{I}}' = \bar{\mathtt{I}}\ )\end{array}}{\mathtt{this} : \langle\ \bar{\mathtt{F}}\ \vert\ \overline{\sigma}\ \rangle \vdash \mathtt{J\,m}\ (\bar{\mathtt{J}}\ \bar{\mathtt{x}})\{\mathtt{return\ e;}\} : \mathtt{J\,m}\ (\bar{\mathtt{J}}) \ \vert\ \langle\ \bar{\mathtt{F}}'\ \vert\ \overline{\sigma}'\ \vert\ \bar{\mathtt{I}}'\ \rangle} \qquad \text{(M-Ok)}$$

Fig. 4 FRJ: Typing rules for expressions and method declarations

Trait expression typing:

$$\frac{\begin{array}{l} mPSig(\overline{\mathtt{S}}) = \overline{\sigma} \qquad mPSig(\mathtt{M}_1 ... \mathtt{M}_p) = \zeta_1 ... \zeta_p \qquad p \geq 0 \\ \forall i \in 1..p, \quad \mathtt{this} : \langle\ \bar{\mathtt{F}}\ ;\ \overline{\sigma} \cdot \zeta_1 ... \zeta_p\ \rangle \vdash \mathtt{M}_i : \mu_i \qquad \mu_i = \zeta_i\ ;\ \langle\ \bar{\mathtt{F}}^{(i)}\ ;\ \overline{\zeta}^{(i)}\ ;\ \bar{\mathtt{I}}^{(i)}\ \rangle \\ \bar{\mathtt{F}} = \cup_{i \in 1..p} \bar{\mathtt{F}}^{(i)} \qquad \overline{\sigma} = discard(names(\zeta_1 ... \zeta_p), (\cup_{i \in 1..p} \overline{\zeta}^{(i)})) \\ \zeta_1 ... \zeta_p \cup (\cup_{i \in 1..p} \overline{\zeta}^{(i)}) \cup mPSig(\cup_{i \in 1..p} \bar{\mathtt{I}}^{(i)})\ \ \mathbf{ok} \end{array}}{\vdash \{\ \bar{\mathtt{F}};\ \bar{\mathtt{S}};\ \mathtt{M}_1 ... \mathtt{M}_p\ \} : \mu_1 ... \mu_p} \quad \text{(T-TEBasic)}$$

$$\frac{\vdash \mathtt{trait\ T}\ \cdots : \bar{\mu}}{\vdash \mathtt{T} : \bar{\mu}} \quad \text{(T-TE)}$$

$$\frac{\begin{array}{l} \vdash \mathtt{TE}_1 : \mu_1 ... \mu_p \qquad \vdash \mathtt{TE}_2 : \mu_{p+1} ... \mu_{p+q} \\ p, q \geq 1 \qquad \forall i \in 1..p+q, \quad \mu_i = \zeta_i\ ;\ \langle\ \bar{\mathtt{F}}^{(i)}\ ;\ \overline{\sigma}^{(i)}\ ;\ \bar{\mathtt{I}}^{(i)}\ \rangle \\ \cup_{i \in 1..p+q} \bar{\mathtt{F}}^{(i)}\ \ \mathbf{ok} \qquad \zeta_1 ... \zeta_{p+q} \cup (\cup_{i \in 1..p+q} \overline{\sigma}^{(i)}) \cup mPSig(\cup_{i \in 1..p+q} \bar{\mathtt{I}}^{(i)})\ \ \mathbf{ok} \end{array}}{\vdash \mathtt{TE}_1 + \mathtt{TE}_2 : \mu_1 ... \mu_{p+q}} \quad \text{(T-TESum)}$$

$$\frac{\vdash \mathtt{TE} : \bar{\mu} \cdot \mu \cdot \bar{\mu}' \qquad names(\mu) = \mathtt{m}}{\vdash \mathtt{TE\ exclude\ m} : \bar{\mu} \cdot \bar{\mu}'} \quad \text{(T-TEEx)}$$

$$\frac{\begin{array}{l} \vdash \mathtt{TE} : \mu_1 ... \mu_n \qquad n \geq p \geq 1 \qquad \forall i \in 1..n, \quad \mu_i = \zeta_i\ ;\ \langle\ \bar{\mathtt{F}}^{(i)}\ ;\ \overline{\sigma}^{(i)}\ ;\ \bar{\mathtt{I}}^{(i)}\ \rangle \\ names(\zeta_p) = \mathtt{m} \qquad \mathtt{m}' \notin names(\zeta_1 ... \zeta_n) \qquad \zeta_p[\mathtt{m}'/\mathtt{m}] \cup (\cup_{i \in 1..n} \overline{\sigma}^{(i)})\ \ \mathbf{ok} \\ \mu = \zeta_p[\mathtt{m}'/\mathtt{m}]\ ;\ \langle\ \bar{\mathtt{F}}^{(p)}\ ;\ \overline{\sigma}^{(p)}\ ;\ \bar{\mathtt{I}}^{(p)}\ \rangle \end{array}}{\vdash \mathtt{TE\ alias\ m\ as\ m}' : \mu_1 ... \mu_n \mu} \quad \text{(T-TEAl)}$$

$$\frac{\begin{array}{l} \vdash \mathtt{TE} : \mu_1 ... \mu_n \qquad n \geq p \geq 1 \qquad \forall i \in 1..n, \quad \mu_i = \zeta_i\ ;\ \langle\ \bar{\mathtt{F}}^{(i)}\ ;\ \overline{\sigma}^{(i)}\ ;\ \bar{\mathtt{I}}^{(i)}\ \rangle \\ names(\zeta_p) = \mathtt{m} \qquad \mathtt{m}' \notin names(\zeta_1 ... \zeta_n) \qquad \zeta_p[\mathtt{m}'/\mathtt{m}] \cup (\cup_{i \in 1..n} \overline{\sigma}^{(i)})\ \ \mathbf{ok} \\ \mu = \zeta_p[\mathtt{m}'/\mathtt{m}]\ ;\ \langle\ \bar{\mathtt{F}}^{(p)}\ ;\ \overline{\sigma}^{(p)}[\mathtt{m}'/\mathtt{m}]\ ;\ \bar{\mathtt{I}}^{(p)}\ \rangle \end{array}}{\vdash \mathtt{TE\ duplicate\ m\ as\ m}' : \mu_1 ... \mu_n \mu} \quad \text{(T-TEDu)}$$

$$\frac{\begin{array}{l} \vdash \mathtt{TE} : \mu_1 ... \mu_n \qquad n \geq 1 \qquad \forall i \in 1..n, \quad \mu_i = \zeta_i\ ;\ \langle\ \bar{\mathtt{F}}^{(i)}\ ;\ \overline{\sigma}^{(i)}\ ;\ \bar{\mathtt{I}}^{(i)}\ \rangle \\ \overline{\zeta} = \zeta_1 ... \zeta_n \qquad \overline{\sigma} = \overline{\sigma}^{(1)} \cup ... \cup \sigma^{(n)} \qquad \mathtt{m} \in names(\overline{\zeta} \cup \overline{\sigma}) \qquad \mathtt{m}' \notin names(\overline{\zeta}) \\ (\overline{\zeta} \cup \overline{\sigma})[\mathtt{m}/\mathtt{m}'] \cup mPSig(\cup_{i \in 1..n} \bar{\mathtt{I}}^{(i)})\ \ \mathbf{ok} \\ \forall i \in 1..n, \quad \mu_i' = \zeta_i[\mathtt{m}'/\mathtt{m}]\ ;\ \langle\ \bar{\mathtt{F}}^{(i)}\ ;\ \overline{\sigma}^{(i)}[\mathtt{m}'/\mathtt{m}]\ ;\ \bar{\mathtt{I}}^{(i)}\ \rangle \end{array}}{\vdash \mathtt{TE\ rename\ m\ to\ m}' : \mu_1' ... \mu_n'} \quad \text{(T-TEReM)}$$

$$\frac{\begin{array}{l} \vdash \mathtt{TE} : \mu_1 ... \mu_n \qquad n \geq 1 \qquad \forall i \in 1..n, \quad \mu_i = \zeta_i\ ;\ \langle\ \bar{\mathtt{F}}^{(i)}\ ;\ \overline{\sigma}^{(i)}\ ;\ \bar{\mathtt{I}}^{(i)}\ \rangle \\ \bar{\mathtt{F}} = \bar{\mathtt{F}}^{(1)} \cup ... \cup \bar{\mathtt{F}}^{(n)} \qquad \mathtt{f} \in names(\bar{\mathtt{F}}) \qquad \bar{\mathtt{F}}[\mathtt{f}'/\mathtt{f}]\ \ \mathbf{ok} \\ \forall i \in 1..n, \quad \mu_i' = \zeta_i\ ;\ \langle\ \bar{\mathtt{F}}^{(i)}[\mathtt{f}'/\mathtt{f}]\ ;\ \overline{\sigma}^{(i)}\ ;\ \bar{\mathtt{I}}^{(i)}\ \rangle \end{array}}{\vdash \mathtt{TE\ rename\ f\ to\ f}' : \mu_1' ... \mu_n'} \quad \text{(T-TEReF)}$$

Trait declaration typing:

$$\frac{\vdash \mathtt{TE} : \bar{\mu}}{\vdash \mathtt{trait\ T\ is\ TE} : \bar{\mu}} \quad \text{(T-Ok)}$$

Fig. 5 FRJ: Typing rules for trait expressions and trait declarations

Interface declaration typing:

$$\frac{mPSig(\texttt{I})\ \textbf{ok}}{\vdash\ \texttt{interface I extends}\ \bar{\texttt{J}}\ \{\ \overline{\texttt{S}}\ \}\quad \text{OK}}\quad (\text{I-OK})$$

Fig. 6 FRJ: Typing rule for interface declarations

Class declaration typing:

$$\frac{\begin{array}{l}\vdash \texttt{TE} : \mu_1 ... \mu_p \qquad p \geq 0 \qquad \forall i \in 1..p, \quad \mu_i = \zeta_i \ \shortmid\ \langle\ \bar{\texttt{F}}^{(i)}\ \shortmid\ \overline{\sigma}^{(i)}\ \shortmid\ \bar{\texttt{I}}^{(i)}\ \rangle \\ \cup_{i\in 1..p}\bar{\texttt{F}}^{(i)} = \bar{\texttt{J}}\ \bar{\texttt{g}} \qquad \zeta_1 ... \zeta_p \supseteq ((\cup_{i\in 1..p}\overline{\sigma}^{(i)}) \cup mPSig(\bar{\texttt{I}})) \\ \forall \texttt{I}' \in \cup_{i\in 1..p}\bar{\texttt{I}}^{(i)},\ \exists \texttt{I} \in \bar{\texttt{I}},\quad \texttt{I} <: \texttt{I}'\end{array}}{\vdash\ \texttt{class C implements}\ \bar{\texttt{I}}\ \texttt{by TE}\ \{\ \bar{\texttt{J}}\ \bar{\texttt{g}};\ \ \texttt{C}(\bar{\texttt{J}}\ \bar{\texttt{g}})\ \{\ \texttt{this}.\bar{\texttt{g}} = \bar{\texttt{g}};\ \}\ \}\quad \text{OK}}\quad (\text{C-OK})$$

Fig. 7 FRJ: Typing rule for class declarations

6 FFRJ Reduction Rules

The FFRJ reduction rules are given in Fig. 8 (the auxiliary functions *fields*, *methods* and *interfaces* are given in Fig. 3).

Evaluation contexts and redexes:

$$\begin{array}{rcl} E & ::= & [\,] \ \mid\ E.\texttt{f} \ \mid\ E.\texttt{m}(\bar{\texttt{e}}) \ \mid\ \texttt{v.m}(\bar{\texttt{v}}, E, \bar{\texttt{e}}) \ \mid\ (\texttt{I})E \ \mid\ \texttt{new C}(\bar{\texttt{v}}, E, \bar{\texttt{e}}) \\ r & ::= & (\texttt{new C}(\bar{\texttt{v}}))\texttt{.f} \ \mid\ (\texttt{new C}(\bar{\texttt{v}}))\texttt{.m}(\bar{\texttt{v}}) \ \mid\ (\texttt{I})(\texttt{new C}(\bar{\texttt{e}})) \end{array}$$

Reduction rules:

$$\frac{fields(\texttt{C}) = \texttt{I}_1\ \texttt{f}_1; ...; \texttt{I}_n\ \texttt{f}_n}{E[(\texttt{new C}(\texttt{v}_1, ..., \texttt{v}_n))\texttt{.f}_i] \rightarrow E[\texttt{v}_i]}\quad (\text{R-Field})$$

$$\frac{\texttt{I m}\ (\bar{\texttt{I}}\ \bar{\texttt{x}})\ \{\texttt{return e};\ \} \in methods(\texttt{C})}{E[(\texttt{new C}(\bar{\texttt{v}}))\texttt{.m}(\bar{\texttt{u}})] \rightarrow E[\texttt{e}[\bar{\texttt{u}}/\bar{\texttt{x}}, \texttt{new C}(\bar{\texttt{v}})/\texttt{this}]]}\quad (\text{R-Invk})$$

$$\frac{\exists \texttt{J} \in interfaces(\texttt{C}),\quad \texttt{J} <: \texttt{I}}{E[(\texttt{I})(\texttt{new C}(\bar{\texttt{e}}))] \rightarrow E[\texttt{new C}(\bar{\texttt{e}})]}\quad (\text{R-Cast})$$

Fig. 8 FFRJ: Reduction rules