IRIS AperTO

UNIVERSITÀ DEGLI STUDI DI TORINO

AperTO - Archivio Istituzionale Open Access dell'Università di Torino

Separating Type, Behavior, and State to Achieve Very Fine-grained Reuse

This is the author's manuscript

Original Citation:

Availability:

This version is available http://hdl.handle.net/2318/29076 since

Publisher:

John Boyland

Terms of use:

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

06 October 2026

Separating Type, Behavior, and State to Achieve Very Fine-grained Reuse *

Viviana Bono[1], Ferruccio Damiani[1], and Elena Giachino[1]

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

Abstract. Recently, Schärli et al. pointed out that 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*. To overcome this problem, Schärli et al. proposed *traits*, which are composable pure units of behavior reuse consisting only of methods. 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, traits live together with the traditional class-based inheritance. Therefore, besides their primary role of generators of instances, classes can still play a secondary role of units of (state and behavior) reuse, and a style of programming oriented to reuse is not enforced by the language, but left to the programmer's skills. 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, class-based statically typed programming languages should be designed according to the principle that *each programming construct must have exactly one role*. We present language constructs that separate completely the declarations of object *type*, *behavior*, *state*, and *generator*.

Keywords. Inheritance, Trait, Subtyping, Flattening.

1 Introduction

It is common opinion that standard class-based inheritance does not support low coupling and, therefore, does not support 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 [24]. 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 [16], 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. [29, 12], by pointing out that both single and multiple class-based inheritance are often

* Work partially supported by MIUR Cofin'06 EOS DUE project. The funding body is not responsible for any use that might be made of the results presented here.

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* (hence it should provide a *minimal* set of sensibly reusable features). Schärli et al. also observed that mixins [9, 21, 15, 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.

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. 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, traits live together with the traditional class-based inheritance. Therefore, besides their primary role of generators of instances, classes can still play a secondary role of units of (state and behavior) 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 [25] for a brief overview). In some of these proposals (notably in the Scala language [26]) 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 rules out operations on traits such as method aliasing and exclusion, 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 [30] (see also Cook et al. [10]).

Having in mind the need of promoting interface-based polymorphism and arbitrary composable units of behavior 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, class-based statically typed programming languages should be designed according to the principle that *each programming construct must have exactly one role*. We propose programming language features that separate completely the declarations of object *type*, *behavior*, *state*, and *generator*, namely, we consider:

- *Interfaces*, as pure types.
- *Traits*, as pure units of behavior reuse.
- *Records*, as pure units of state definition and initialization reuse.
- *Classes*, as pure generators of instances.

Interfaces, traits, and records can be defined by composing other interfaces, traits, and records respectively. Classes are defined by composing records, traits, interfaces and by adding glue fields and methods. Note that classes cannot be reused, and in particular 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.

Another outcome of the complete role separation is that multiple inheritance is subsumed by ensuring that, in the spirit of the trait proposal [12], the composite unit has complete control over the composition and must resolve conflicts explicitly. Multiple inheritance with respect to methods is obtained via the trait construct, and multiple inheritance with respect to fields is obtained via our novel record construct.

The paper is organized as follows. Section 2 illustrates our proposal by presenting an example in a JAVA-like syntax. Section 3 discusses briefly the issue of giving a clean semantics to reusable components and illustrates the syntax of a core calculus for very fine-grained reuse based on the constructs introduced above, outlines its type system, gives a semantics by translation, and states a type soundness result. We conclude by discussing some related work.

2 An example

In order to present an introductory example, we exploit a JAVA-like syntax (at present, we do not have a prototypical language implementing our proposal). In a class the only (implicitly) public methods are those declared in 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`, `String`, etc.) we assume an interface, a trait and a record. The same name can be used to denote both the interface, the record, the trait, and the class (the complete separation of roles ensures that the names of different components cannot occur in the same place of a program). The `Object` interface is implicity extended by any other interface, and the `Object` trait and record are implicity used by any class.

We consider the problem of building a software system for a non-profit organization promoting car-sharing, that is, there are participants that offers car lifts to other participants, and the main task of the association is to allow the participants offering/requesting lifts to find the best match for their needs. The specifications for this software are longer and more complicated than that, but for our purpose (and due to the lack of space) we will limit ourselves to consider only the ones described and we model only a limited version of users of such an organization.

We start from defining some types, that in our proposal are represented by interfaces:

```
interface CarMatcherType {...}
interface CarOwnerType extends CarMatcherType {...}
```

Then we build our state description through the record `MemberFeatures`:

```
record MemberFeatures {
  String name; String surname; String address;
```

```
  MemberFeatures(String name, String surname, String address) {
    this.name = name;
    this.surname = surname;
    this.address = address;
  }
}
```

and the records `InsuranceFeatures` and `CarFeatures` that are defined in a similar way. Note that records declares fields and constructors, therefore they are unit of reuse of fields and their initialization code.

Having interfaces and records, we can compose our object generators, that is, building some of the classes:

```
class CarPassenger implements CarMatcherType
                   uses MemberFeatures, InsuranceFeatures {
    ... /* accessory methods */
    CarPassenger(String name, Sting surname, String address,
                 String company, String kind, String expirationDate) {
       MemberFeatures(name, surname, address);
       InsuranceFeatures(company, kind, expirationDate);
    }
}

class CarOwner implements CarOwnerType
               uses MemberFeatures, InsuranceFeatures, CarFeatures {
    ... /* accessory methods */
    CarOwner(String name, Sting surname, String address,
             String company, String kind, String expirationDate,
             String make, String numberPlate, int seatNumber) {
       MemberFeatures(name, surname, address);
       InsuranceFeatures(company, kind, expirationDate);
       CarFeatures(make, numberPlate, seatNumber)
  }
}
```

Besides the two interfaces `CarMatcherType` and `CarOwnerType` presented above, we assume an interface `CarMatcherManagerType` that represents the type for operations on tables of car-sharers. We assume also the existence of a data base, that stores the data of the organization, and we define a trait `DbHandler` that provides the methods that map the tables to the appropriate relations in the data base:

```
trait DbHandler requires {Map table;} {
  ...
}
```

The class `CarPassengerManager` contains some methods: one for the insertion of a car-sharer in a `Map` table (`Map` is the interface belonging to the Java API's); one that, given a criterium ("name", "surname", or any other field name of `MemberFeatures` or `InsuranceFeatures`) performs an extraction of an entry from the table; and one that, given a criterium, deletes an entry from the table

(it returns true if car passenger(s)/owner(s) matching the criterium are found and deleted, false otherwise). Moreover, it contains the methods `updateDB` and `loadDB`, that use the methods of the trait `DbHandler`.

```
class CarPassengerManager implements CarMatcherManagerType
                          uses DbHandler {

  Map passengerTable;

  ... /* constructors and accessory methods */

  void insert(CarMatcherType c) { ... }
  CarMatcherType search(String criterium, String data) { ... }
  boolean delete(String criterium, String data) { ... }
  void updateDB() { ... }
  void loadDB() { ... }
}
```

The classes `CarOwnerOperations` and `CarOwnerPassengerOperations` are defined similarly:

```
class CarOwnerManager implements CarMatcherManagerType
                      uses DbHandler {

  Map ownerTable;

  ...
}

/* CarOwnerPassengerManager is the class of whom both offers and asks */
class CarOwnerPassengerManager implements CarMatcherManagerType
                               uses DbHandler {

  Map ownerPassengerTable;

  ...
}
```

Note the way the code reuse (that is, the interface, record, and trait reuse) is realized in the class compositions.

3 FCJ: a Calculus for Reusable Components

Supplying a clean semantics to reusable components is traditionally a critical issue. We argue that the semantics of reusable components in object oriented class based programming languages should be defined according to the following principle: *the semantics of a class member introduced through a reusable com-*

ponent should be identical to the semantics of the same member defined directly within a class.[1]

The above principle is been inspired by:

- the claim that "the semantics of overloading and inheritance is clean only if it can be understood through a copy semantics, whereby programs are transformed to equivalent programs without subclasses, and the effect of inheritance is obtained through copying" [4];
- the *copy principle* for mixins [3];
- the *flattening property* for traits [12];
- the proposal that "any type system that accommodates traits should have the property that programs should be equivalent to their flattened counterparts" [25].

To provide a formal account of our idea, we present FCJ (Featherweight-Compositional Java), a core calculus for interfaces, traits, records, and classes, inspired by FJ (Featherweight Java) [19] and FTJI (Featherweight-Trait Java with Interfaces) [25].

3.1 Syntax

The syntax of our calculus, FCJ, is presented in Figure 1. We also consider a calculus, FFCJ (Flat FCJ), obtained by removing the portions of the syntax highlighted in grey. The calculus FFCJ can be considered a subset of FJI (FJ with Interfaces)[2]. Indeed, FFCJ is not a proper subset of FJI: its syntax for class constructors is more liberal, in order to provide enough expressivity to make the flattening of records possible. Besides the differences in the class constructors, the main difference between FCJ and FTJI are the following:

- Classes and traits are not types, and class-based inheritance is not present;
- Traits can directly access fields and must declare the type of the required fields and methods.[3]
- 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). This requirement might be relaxed by saying that two methods with the same name do not conflict if they originate from the same subtrait (as in [22]).
- The *override* operation (that layers additional methods over an existing trait) is not present.
- We have the operations *hide* (that forms a new trait by hiding a method name from an existing trait, thus permanently binding the method to the trait) and *rename* (that creates a new trait by renaming all the occurrences of a method name from an existing trait).[4]

[1] Notice that this principle just aims to provide a canonical semantics to reusable components, it is not an especially effective implementation technique.

[2] This calculus was introduced in [25] to state the flattening property for FTJI.

[3] Field requirements were introduced in [14].

[4] Method hiding and renaming were introduced in [27].

$$
\begin{array}{l}
\texttt{ID} ::= \texttt{interface I extends } \overline{\texttt{I}} \texttt{ \{ } \overline{\texttt{S}}\texttt{; \}} \\
\texttt{S} \;\; ::= \texttt{I m (}\overline{\texttt{I}}\ \overline{\texttt{x}}\texttt{)} \\
\\
\texttt{TD} \;\; ::= \;\; \texttt{trait T requires \{}\overline{\texttt{F}}\texttt{; }\overline{\texttt{S}}\texttt{;\} uses } \overline{\texttt{TE}} \texttt{ \{ } \overline{\texttt{M}} \texttt{ \}} \\
\texttt{F} \;\; ::= \texttt{I f} \\
\texttt{TE} \;\; ::= \;\; \texttt{T} \mid \texttt{TE exclude m} \mid \texttt{TE alias m as m} \mid \texttt{TE hide m} \\
\qquad \mid \texttt{TE rename m to m} \mid \texttt{TE rename f to f} \\
\texttt{M} \;\; ::= \texttt{S \{ return e; \}} \\
\texttt{e} \;\; ::= \texttt{x} \mid \texttt{this.f} \mid \texttt{e.m(}\overline{\texttt{e}}\texttt{)} \mid \texttt{new C(}\overline{\texttt{e}}\texttt{)} \mid \texttt{(I)e} \\
\\
\texttt{RD} \;\; ::= \;\; \texttt{record R uses } \overline{\texttt{RE}} \texttt{ \{ } \overline{\texttt{F}}\texttt{; KR \}} \\
\texttt{RE} \;\; ::= \;\; \texttt{R} \mid \texttt{RE rename f to f} \\
\texttt{KR} \;\; ::= \;\; \texttt{R(}\overline{\texttt{I}}\ \overline{\texttt{x}}\texttt{) \{ R(}\overline{\texttt{e}}\texttt{); ...; R(}\overline{\texttt{e}}\texttt{); this.}\overline{\texttt{f}} = \overline{\texttt{e}}\texttt{; \}} \\
\\
\texttt{CD} ::= \texttt{class C implements } \overline{\texttt{I}} \texttt{ uses } \overline{\texttt{RE}} \texttt{ and } \overline{\texttt{TE}} \texttt{ \{ } \overline{\texttt{F}}\texttt{; KC; } \overline{\texttt{M}} \texttt{ \}} \\
\texttt{KC} ::= \texttt{C(}\overline{\texttt{I}}\ \overline{\texttt{x}}\texttt{) \{ } \overline{\texttt{I}}\ \overline{\texttt{x}} \texttt{ = e; R(}\overline{\texttt{e}}\texttt{); ...R(}\overline{\texttt{e}}\texttt{); this.}\overline{\texttt{f}} = \overline{\texttt{e}}\texttt{; \}}
\end{array}
$$

Fig. 1. FCJ: Syntax

- Note that the operations *exclude* (that forms a new trait by removing a method from an existing trait) and *alias* (that forms a new trait by adding a new name for an existing method) correspond exactly to the `minus` and `with` operators of FTJI, respectively.[5]
- We have records and record expressions with the operations of *symmetric sum* (that merges two disjoint records to form a new record) and *field renaming* (that forms a new record by renaming all the occurrences of a field name from an existing record).

We use the overbar sequence notation according to [19]. For instance: "$\overline{\texttt{f}}$" denotes the possibly empty sequence "$\texttt{f}_1, ..., \texttt{f}_n$", the pair "$\overline{\texttt{I}}\ \overline{\texttt{x}}$" stands for "$\texttt{I}_1\ \texttt{x}_1, ..., \texttt{I}_n\ \texttt{x}_n$", "$\overline{\texttt{I}}\ \overline{\texttt{x}}$;" stands for "$\texttt{I}_1\ \texttt{x}_1; ...; \texttt{I}_n\ \texttt{x}_n$;", the assignment "$\texttt{this.}\overline{\texttt{f}} = \overline{\texttt{e}}$;" stands for "$\texttt{this.f}_1 = \texttt{e}_1; ...; \texttt{this.f}_n = \texttt{e}_n$;", and "$\overline{\texttt{I}}\ \overline{\texttt{x}} = \overline{\texttt{e}}$;" stands for the sequence of local (to the body of a class constructor) variable declarations/initializations "$\texttt{I}_1\ \texttt{x}_1 = \texttt{e}_1; ...; \texttt{I}_n\ \texttt{x}_n = \texttt{e}_n$;". The empty sequence is denoted by "$\bullet$".

Sequences of named elements (e.g., methods signatures) are assumed to contain no duplicate names, the sequence of the names of the elements of $\overline{\texttt{S}}$ is denoted by *names*($\overline{\texttt{S}}$), the subsequence of the elements of $\overline{\texttt{S}}$ with the names $\overline{\texttt{n}}$ is denoted by *extract*($\overline{\texttt{n}}, \overline{\texttt{S}}$), and *discard*($\overline{\texttt{n}}, \overline{\texttt{S}}$) denotes the sequence obtained from $\overline{\texttt{S}}$ by removing the elements with the names $\overline{\texttt{n}}$. Following [19], we use set-based notation for operators over sequences of named elements. For instance, $\texttt{M} = \texttt{I m}\,(\overline{\texttt{I}}\ \texttt{x}) \in \overline{\texttt{M}}$ means that the method declaration `M` occurs in $\overline{\texttt{M}}$. In the union and in the intersection of sequences of named elements, denoted by $\overline{\texttt{S}} \cup \overline{\texttt{Z}}$ and

[5] When a recursive method is aliased, its recursive invocation refers to the original method. Another interpretation of aliasing (see [22]) ensures that the recursive invocations refer to the alias.

$\overline{\mathtt{S}} \cap \overline{\mathtt{Z}}$, 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 CT is a map from class names to class declarations. Similarly, an interface table IT, a trait table TT, and a record table RT map interface, trait, and record names to interface, trait, and record declarations, respectively. A FCJ program is a 5-tuple $(\mathtt{IT}, \mathtt{TT}, \mathtt{RT}, \mathtt{CT}, \mathtt{e})$. In presenting the type system and the flattening translation we assume fixed, global tables IT, TT, RT, and CT. We also assume that these tables are *well-formed*, i.e., they contain an entry for each interface/trait/record/class mentioned in the program, and the interface subtyping, trait reuse, and record reuse graphs are acyclic.

3.2 Typing

The FCJ type system combines nominal and structural typing. It typechecks the uses of method parameters 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, within a:

- trait definition, takes into account the field and methods *required* by the trait and the methods *provided* by the trait (i.e., either defined directly within the trait or introduced through used traits);
- record definition, takes into account the fields *provided* by the record (i.e., either defined directly within the record or introduced through used records);
- class definition, takes into account the fields and methods of the class (i.e., either defined directly within the class or introduced through used records and traits, respectively).

The body of a method is type-checked by assuming an interface name as type for each parameter of the method and a pair $\langle\ \overline{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle$ as type for the this metavariable. The type $\langle\ \overline{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle$, where $\overline{\mathtt{F}} = \mathtt{I}_1\,\mathtt{f}_1 ... \mathtt{I}_p\,\mathtt{f}_p$ $(p \geq 0)$ and $\overline{\sigma} = \mathtt{J}_1\mathtt{m}_1(\overline{\mathtt{J}}^{(1)}) ... \mathtt{J}_q\mathtt{m}_q(\overline{\mathtt{J}}^{(q)})$ $(q \geq 0)$, specifies that this has the fields $\overline{\mathtt{F}}$ and methods with signatures $\overline{\sigma}$.

The type of an object creation expression $\mathtt{new}\,\mathtt{C}(\cdots)$ is the conjunction $\mathtt{I}_1 \wedge ... \wedge \mathtt{I}_n$ of the interfaces implemented by the class C.[6] We will write $\bigwedge \overline{\mathtt{I}}$, where $\overline{\mathtt{I}}$ is a sequence of $n \geq 0$ interfaces $\mathtt{I}_1, ..., \mathtt{I}_n$, to denote the type $\mathtt{I}_1 \wedge ... \wedge \mathtt{I}_n$. We identify the empty conjunction $\bigwedge \bullet$ with the empty sequence $\bullet$ and identify the singleton conjunction $\bigwedge \mathtt{I}$ with the interface I.

The type of any other expressions e (not of the form this or $\mathtt{new} \cdots (\cdots)$) is an interface name.

[6] *Conjunctive* (or *intersection*) types have been introduced by Coppo and Dezani [11] (see also [5]) in the contex of functional programming. Recently, Igarashi and Nagira [18] proposed a use of the dual notion of *disjunctive* (or *union*) types for object-oriented programming.

Besides assigning to each expression $\mathtt{e}$ a type describing the object yielded by the evaluation of $\mathtt{e}$, the FCJ type system assigns to $\mathtt{e}$ also an effect. An effect is a triple

$$\langle\ \bar{\mathtt{F}}\ \mid\ \overline{\sigma}\ \mid\ \bar{\mathtt{I}}\ \rangle$$

specifying that the expression $\mathtt{e}$ selects the fields $\bar{\mathtt{F}}$ and the methods $\overline{\sigma}$ on $\mathtt{this}$, and requires that $\mathtt{this}$ implements the interfaces $\bar{\mathtt{I}}$.

The typing rules for interface declarations, expressions, method declarations, trait declarations, record declarations, and class declarations are syntax directed, with one rule for each form of term, except that (following [19]) there are three rules for casts. The most important typing judgements are the following:

- $\boxed{\vdash \mathtt{interface\ I\ extends}\ \bar{\mathtt{I}}\ \{\ \bar{\mathtt{S}};\ \}\quad \mathrm{OK}}$
  To be read: the declaration of the interface $\mathtt{I}$ is well-typed.
- $\boxed{\bar{\mathtt{x}} : \bar{\mathtt{I}},\ \mathtt{this} : \langle\ \bar{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle \vdash \mathtt{e} : \eta\ \mid\ \delta}$ where

$$\eta = \begin{cases} \langle\ \bar{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle & \text{if } \mathtt{e} = \mathtt{this} \\ \bigwedge \bar{\mathtt{J}} & \text{if } \mathtt{e} = \mathtt{new\,C}(\cdots) \text{ for some } \mathtt{C} \text{ such that} \\ & \quad \mathtt{CT(C)} = \mathtt{class\,C\ implements}\ \bar{\mathtt{J}}\ \mathtt{uses} \cdots \\ \mathtt{I} & \text{otherwise (for some interface } \mathtt{I}) \end{cases}$$
$$\delta = \langle\ \bar{\mathtt{G}}\ \mid\ \overline{\zeta}\ \mid\ \bar{\mathtt{J}}'\ \rangle$$

  To be read: under the assumption that the variables $\bar{\mathtt{x}}$ have types $\bar{\mathtt{I}}$ and that the metavariable $\mathtt{this}$ has fields $\bar{\mathtt{F}}$ and methods $\overline{\sigma}$, the expression $\mathtt{e}$ is well-typed with type η and effect δ.
- $\boxed{\mathtt{this} : \langle\ \bar{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle \vdash \mathtt{I\ m}\ (\bar{\mathtt{I}}\ \bar{\mathtt{x}})\{\mathtt{return\ e};\} : \mu}$ where

$$\mu = \zeta\ \mid\ \delta$$

  To be read: under the assumption that $\mathtt{this}$ has fields $\bar{\mathtt{F}}$ and methods $\overline{\sigma}$, the declaration of method $\mathtt{m}$ is well-typed with type μ. I.e., the method $\mathtt{m}$ has signature ζ and its body has effect δ.
- $\boxed{\vdash \mathtt{trait\ T\ requires}\ \{\bar{\mathtt{F}};\ \bar{\mathtt{S}};\}\ \mathtt{uses}\ \overline{\mathtt{TE}}\ \{\bar{\mathtt{M}}\} : \bar{\mu}}$ where

$$\bar{\mu} = \mu_1 ... \mu_n \qquad (n \geq 0)$$

  To be read: the declaration of trait $\mathtt{T}$ is well-typed with type $\bar{\mu}$. I.e., the trait $\mathtt{T}$ provides n methods with types $\mu_1, ..., \mu_n$, respectively.
- $\boxed{\vdash \mathtt{record\ R\ uses}\ \overline{\mathtt{RE}}\ \{\ \bar{\mathtt{F}};\ \mathtt{KR}\ \} : \rho}$ where

$$\rho = \mathtt{R}(\bar{\mathtt{I}}) \mid \langle\ \bar{\mathtt{G}}\ \rangle$$

  To be read: the declaration of record $\mathtt{R}$ is well-typed with type ρ. I.e., the record $\mathtt{R}$ provides a field initializer with signature $\mathtt{R}(\bar{\mathtt{I}})$ and the fields $\bar{\mathtt{G}}$.
- $\boxed{\vdash \mathtt{class\ C\ implements}\ \bar{\mathtt{I}}\ \mathtt{uses}\ \overline{\mathtt{RE}}\ \mathtt{and}\ \overline{\mathtt{TE}}\ \{\ \bar{\mathtt{F}};\ \mathtt{KC}\ \bar{\mathtt{M}};\ \}\quad \mathrm{OK}}$
  To be read: the declaration of the class $\mathtt{C}$ is well-typed.

– $\boxed{\vdash_{\mathrm{FCJ}} (\mathtt{IT}, \mathtt{TT}, \mathtt{RT}, \mathtt{CT}, \mathtt{e}) : \bigwedge \bar{\mathtt{I}}}$

To be read: the program $(\mathtt{IT}, \mathtt{TT}, \mathtt{RT}, \mathtt{CT}, \mathtt{e})$ has type $\bigwedge \bar{\mathtt{I}}$ w.r.t. the $\vdash_{\mathrm{FCJ}}$ system. I.e., the interfaces in IT, the records in RT, the traits in TT, and the classes in CT are well-typed, and the expression e is well-typed with type $\bigwedge \bar{\mathtt{I}}$ and empty effect (i.e., $\langle \bullet \mid \bullet \mid \bullet \rangle$) under the empty set of assumptions.

3.3 Semantics by Translation and Type Soundness

A FFCJ program is a FCJ program with empty traits and record tables. The flattening translation removes the trait and the record tables and replaces the class table with a suitable one containing only FFCJ classes. The translation is specified through the function $[\![\cdot]\!]$, given in Figure 2, that maps a FCJ class declaration to a FFCJ class declaration. For the sake of simplicity, we assume that the names of the parameters of the record constructors and the names of the parameters of the class constructors are all distinct.

A FJI program is a 3-tuple $(\mathtt{IT}, \mathtt{CT}, \mathtt{e})$. Let $\vdash_{\mathrm{FJI}}$ denote the obvious extension of the FJ type system [19] to deal with interfaces and richer (as in FFCJ) class constructors. The main typing judgment of the $\vdash_{\mathrm{FJI}}$ type system is

– $\boxed{\vdash_{\mathrm{FJI}} (\mathtt{IT}, \mathtt{CT}, \mathtt{e}) : \nu}$ where

$$\nu \text{ is either a class } \mathtt{C} \text{ or an interface } \mathtt{I}.$$

To be read: the program $(\mathtt{IT}, \mathtt{CT}, \mathtt{e})$ has type ν w.r.t. the $\vdash_{\mathrm{FJI}}$ system. I.e., the interfaces in IT and the classes in CT are well-typed, and the expression e is well-typed with type ν under the empty set of assumptions.

The type soundness result is split in two parts: firstly it says that the flattening translation preserves the type of programs w.r.t. $\vdash_{\mathrm{FJI}}$, then it relates $\vdash_{\mathrm{FCJ}}$ with $\vdash_{\mathrm{FJI}}$.

Theorem 1 (Type Soundness).
If $\vdash_{\mathrm{FCJ}} (\mathtt{IT}, \mathtt{TT}, \mathtt{RT}, \mathtt{CT}, \mathtt{e}) : \bigwedge \bar{\mathtt{I}}$, *then:*

1. $\vdash_{\mathrm{FCJ}} (\mathtt{IT}, \bullet, \bullet, [\![\mathtt{CT}]\!], \mathtt{e}) : \bigwedge \bar{\mathtt{I}}$, *and*
2. $\vdash_{\mathrm{FJI}} (\mathtt{IT}, [\![\mathtt{CT}]\!], \mathtt{e}) : \nu$, *where*

$$\nu = \begin{cases} \mathtt{C} & \textit{if } \mathtt{e} = \mathtt{new}\,\mathtt{C}(\cdots) \textit{ for some } \mathtt{C} \textit{ such that} \\ & \quad \mathtt{CT}(\mathtt{C}) = \mathtt{class}\,\mathtt{C}\,\mathtt{implements}\,\bar{\mathtt{I}}\,\mathtt{uses}\cdots \\ \mathtt{I} & \textit{(with } \mathtt{I} = \bar{\mathtt{I}}\textit{), otherwise.} \end{cases}$$

$$
\left[\!\!\left[\begin{array}{l}
\texttt{class C implements } \bar{\texttt{J}} \\
\texttt{uses } \overline{\texttt{RE}} \texttt{ and } \overline{\texttt{TE}} \texttt{ \{ } \bar{\texttt{F}}\texttt{;} \\
\quad \texttt{C}(\bar{\texttt{I}}\ \bar{\texttt{x}}) \quad \texttt{\{ } \bar{\texttt{I}}'\ \bar{\texttt{x}}' \texttt{ = } \bar{\texttt{e}}'\texttt{;} \\
\quad\quad \texttt{R}_1(\bar{\texttt{e}}^{(1)})\texttt{;} ... \texttt{R}_n(\bar{\texttt{e}}^{(n)})\texttt{;} \\
\quad\quad \texttt{this.}\bar{\texttt{f}} \texttt{ = } \bar{\texttt{e}}\texttt{; \}} \\
\quad \bar{\texttt{M}} \texttt{ \}}
\end{array}\right]\!\!\right]
\stackrel{\text{def}}{=}
\begin{array}{l}
\texttt{class C implements } \bar{\texttt{J}} \\
\texttt{\{ } \bar{\texttt{F}}\texttt{; } [\![\overline{\texttt{RE}}]\!]\texttt{;} \\
\quad \texttt{C}(\bar{\texttt{I}}\ \bar{\texttt{x}})\ \texttt{\{ } \bar{\texttt{I}}'\ \bar{\texttt{x}}' \texttt{ = } \bar{\texttt{e}}'\texttt{;} \\
\quad\quad rPars(\overline{\texttt{RE}}) = \bar{\texttt{e}}^{(1)} ... \bar{\texttt{e}}^{(n)};\ xInit(\overline{\texttt{RE}});\ fInit(\overline{\texttt{RE}}); \\
\quad\quad \texttt{this.}\bar{\texttt{f}} \texttt{ = } \bar{\texttt{e}}\texttt{; \}} \\
\quad [\![\overline{\texttt{TE}}]\!]\ \bar{\texttt{M}}\texttt{\}}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\texttt{T}]\!] & \stackrel{\text{def}}{=} & [\![\overline{\texttt{TE}}]\!]\ \bar{\texttt{M}} \qquad \textbf{if } \texttt{TT(T)} = \texttt{trait T requires } \{\bar{\texttt{F}};\ \bar{\texttt{S}};\} \texttt{ uses } \overline{\texttt{TE}}\ \texttt{\{ } \bar{\texttt{M}}\ \texttt{\}} \\
[\![\texttt{TE exclude m}]\!] & \stackrel{\text{def}}{=} & discard(\texttt{m}, \bar{\texttt{M}}) \qquad \textbf{if } [\![\texttt{TE}]\!] = \bar{\texttt{M}} \text{ and } \texttt{m} \in names(\bar{\texttt{M}}) \\
[\![\texttt{TE alias m as n}]\!] & \stackrel{\text{def}}{=} & \bar{\texttt{M}}\ \texttt{I n}(\bar{\texttt{I}}\ \bar{\texttt{x}})\texttt{\{return e;\}} \qquad \textbf{if } [\![\texttt{TE}]\!] = \bar{\texttt{M}} \text{ and } \texttt{I m}(\bar{\texttt{I}}\ \bar{\texttt{x}})\texttt{\{return e;\}} \in \bar{\texttt{M}} \\
[\![\texttt{TE hide m}]\!] & \stackrel{\text{def}}{=} & [\![\texttt{TE}]\!][\texttt{m}_0/\texttt{m}] \qquad \textbf{with } \texttt{m}_0 \text{ fresh} \\
[\![\texttt{TE rename m to n}]\!] & \stackrel{\text{def}}{=} & [\![\texttt{TE}]\!][\texttt{n}/\texttt{m}] \\
[\![\texttt{TE rename f to g}]\!] & \stackrel{\text{def}}{=} & [\![\texttt{TE}]\!][\texttt{f}/\texttt{g}] \\
[\![\texttt{TE}_1, ..., \texttt{TE}_n]\!] & \stackrel{\text{def}}{=} & [\![\texttt{TE}_1]\!] ... [\![\texttt{TE}_n]\!]
\end{array}
$$

$$
\begin{array}{rcl}
[\![\texttt{R}]\!] & \stackrel{\text{def}}{=} & [\![\overline{\texttt{RE}}]\!]\ \bar{\texttt{F}}; \qquad \textbf{if } \texttt{RT(R)} = \texttt{record R uses } \overline{\texttt{RE}}\ \texttt{\{ } \bar{\texttt{F}};\ \texttt{KR}\ \texttt{\}} \\
[\![\texttt{RE rename f to f}']\!] & \stackrel{\text{def}}{=} & [\![\texttt{RE}]\!][\texttt{f}'/\texttt{f}] \\
[\![\texttt{RE}_1, ..., \texttt{RE}_n]\!] & \stackrel{\text{def}}{=} & [\![\texttt{RE}_1]\!]; ...; [\![\texttt{RE}_n]\!]
\end{array}
$$

where the auxiliary lookup functions $rPars(\cdot)$, $fInit(\cdot)$, and $xInit(\cdot)$ are defined as follows:

$$
\begin{array}{rcl}
rPars(\texttt{R}) & \stackrel{\text{def}}{=} & \bar{\texttt{I}}\ \bar{\texttt{x}}, \qquad \textbf{if } \texttt{TT(R)} = \texttt{record R uses } \cdots\ \texttt{\{ } \cdots\ \texttt{R}(\bar{\texttt{I}}\ \bar{\texttt{x}})\ \{\cdots\}\ \texttt{\}} \\
rPars(\texttt{RE rename f to f}') & \stackrel{\text{def}}{=} & rPars(\texttt{RE}) \\
rPars(\texttt{RE}_1, ..., \texttt{RE}_n) & \stackrel{\text{def}}{=} & rPars(\texttt{RE}_1); ...; rPars(\texttt{RE}_n)
\end{array}
$$

$$
\begin{array}{rcl}
fInit(\texttt{R}) & \stackrel{\text{def}}{=} & fInit(\overline{\texttt{RE}});\ \texttt{this.}\bar{\texttt{f}} = \bar{\texttt{e}}; \\
 & & \textbf{if } \texttt{TT(R)} = \texttt{record R uses } \overline{\texttt{RE}}\ \texttt{\{ } \cdots\ \texttt{this.}\bar{\texttt{f}} = \bar{\texttt{e}}\texttt{;\}\ \}} \\
fInit(\texttt{RE rename f to f}') & \stackrel{\text{def}}{=} & fInit(\texttt{RE})[\texttt{f}'/\texttt{f}] \\
fInit(\texttt{RE}_1, ..., \texttt{RE}_n) & \stackrel{\text{def}}{=} & fInit(\texttt{RE}_1); ...; fInit(\texttt{RE}_n)
\end{array}
$$

$$
\begin{array}{rcl}
xInit(\texttt{R}) & \stackrel{\text{def}}{=} & rPars(\overline{\texttt{RE}}) = \bar{\texttt{e}}^{(1)} ... \bar{\texttt{e}}^{(n)};\ xInit(\overline{\texttt{RE}}) \\
 & & \textbf{if } \texttt{TT(R)} = \texttt{record R uses } \overline{\texttt{RE}}\ \texttt{\{ } \cdots\ \texttt{R}(\bar{\texttt{I}}\ \bar{\texttt{x}})\ \{\texttt{R}_1(\bar{\texttt{e}}^{(1)}); ...; \texttt{R}_n(\bar{\texttt{e}}^{(n)});\ \cdots\}\ \texttt{\}} \\
xInit(\texttt{RE rename f to f}') & \stackrel{\text{def}}{=} & xInit(\texttt{RE})[\texttt{f}'/\texttt{f}] \\
xInit(\texttt{RE}_1, ..., \texttt{RE}_n) & \stackrel{\text{def}}{=} & xInit(\texttt{RE}_1); ...; xInit(\texttt{RE}_n)
\end{array}
$$

Fig. 2. Flattening FCJ to FFCJ

Example 2. As an example of the translation let consider an implementation of the class `ColorPoint`.

```
interface IPoint { String toString();  Integer getX();  Integer getY(); }

interface IColor { String toString();  String getColor(); }
```

```
interface IColorPoint extends IColor,IPoint { }

record RPoint { Integer x;  Integer y;
                RPoint(Integer x, Integer y) { this.x=x; this.y=y; }
}

record RColor { String color;
                RColor(String color) { this.color=color; }
}

trait TPoint requires { Integer x; Integer y; } {
   Integer getX() { return x; }
   Integer getY() { return y; }
   String toString() { return x + ``,'' + y; }
}

trait TColor requires { String color; } {
   String getColor() { return color; }
   String toString() { return color; }
}

class ColorPoint implements IColorPoint
                uses RPoint,RColor
                and (TPoint rename toString to toStringP),
                    (TColor rename toString to toStringC) {
   ColorPoint(Integer px, Integer py, String pcolor) {
      RPoint(px,py); RColor(pcolor);
   }
   String toString() { return this.toStringP() + ``,'' + this.toStringC(); }
}
```

The FCJ code translated in FJI is as follows, where the interface `IPoint`, `IColor` and `IColorPoint` are as before:

```
class ColorPoint implements IColorPoint {
   Integer x;
   Integer y;
   String color;
   ColorPoint(Integer px, Integer py, String pcolor) {
      Integer x=px;  Integer y=py;  Integer color=pcolor;
      this.x=x;  this.y=y;  this.color=color;
   }
   Integer getX() { return x; }
   Integer getY() { return y; }
   String getColor() { return color; }
   String toStringP() { return x + ``,'' + y; }
   String toStringC() { return color; }
   String toString() { return this.toStringP() + ``,'' +  this.toStringC(); }
}
```

4 Related Work and Conclusions

We attacked the root of the general problem of competing roles played by the same construct, which limits the reuse potential of program components written in mainstream object oriented class based programming languages.[7] 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. [29, 12]. Also the roles of *unit of reuse* and *type* are competing (see Section 1).

We introduced a calculus of interfaces, traits, and records as programming constructs for composing incrementally reusable components from smaller components, and building classes from reusable components. A distinguished feature of our approach is that each of those constructs has exactly one role.[8]

With the aim of achieving flexible typing of reusable components, we have developed a hybrid nominal/structural type system. Further work is necessary to extend this system to deal with generic types. Notably, sophisticated hybrid nominal/structural type systems have been already proposed [13, 23, 28].

Recently, Bergel et al. [7] pointed out several limitations of the trait model. In order to overcome these limitations, they propose 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. Although further work is needed to compare our proposal with stateful traits, we believe that our proposal provides an alternative solution to the limitations of the stateless trait model. In particular, Bergel et al. argued 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 [8] design.

Our proposal addresses exactly this question, by outlining a programming language that ensures that code is structured in composable "single-role" units of reuse.

A special form of reuse is at the base of the contemporary *agile* software development methodologies [2]. Such methodologies range from using the waterfall model on a small scale, that is, repeating the entire waterfall cycle in every iteration (examples of this are variations of the flexible Unified Process, UP [20]), to the use of Extreme Programming [6], where team members work on activities simultaneously. Considering the iterative approach, 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*

[7] Our presentation focuses on statically typed (JAVA-like) languages. However, our proposal may apply also to dynamically typed (SMALLTALK-like) languages.

[8] In the context of dynamically typed languages, Python [1] provides some features that go in this direction.

the software produced in the previous ones. For instance, Holub [17] observes that:

> At the core of the contemporary Agile development methodologies is the concept of parallel design and development. You start programming before you fully specify the program. This technique flies in the face of traditional wisdom — that a design should be complete before programming starts — but many successful projects have proven that you can develop high-quality code more rapidly (and cost effectively) this way than with the traditional pipelined approach. At the core of parallel development, however, is the notion of flexibility. You have to write your code in such a way that you can incorporate newly discovered requirements into the existing code as painlessly as possible.

We believe that an interesting future research direction is to investigate whether our proposal for fine-grained reuse may help in writing software following an agile methodology.

Acknowledgements. We thank Davide Ancona, Paola Giannini, Oscar Nierstrasz, Betti Vennneri, Elena Zucca for useful discussions on the subject of this paper, and Alberto Scotto, for pointing out Holub's articles. We also thank the anonymous FTfJP'07 referees for comments, suggestions, and bibliographic references.

References

1. Python Programming Language – Official Website. http://www.python.org.
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. D. Ancona, E. Zucca, and S. Drossopoulou. Overloading and inheritance. In *FOOL 2001*, January 2001.
5. H. P. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. of Symbolic Logic*, 48:931–940, 1983.
6. K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999.
7. 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.
8. G. Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.
9. 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.
10. 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.
11. M. Coppo and M. Dezani-Ciancaglini. A new type-assignment for λ-terms. *Archiv Math. Logik*, 19:139–156, 1978.

12. 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.
13. K. Fisher and J. Reppy. Inheritance-based subtyping. *Information and Computation*, 177(1):28–55, 2002.
14. Kathleen Fisher and John Reppy. A typed calculus of traits. In *Electronic proceedings of FOOL 2004*, 2004.
15. 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.
16. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
17. A. Holub. Why extends is evil – Improve your code by replacing concrete base classes with interfaces. *JavaWorld.com*, 08/01/03, 2003.
18. A. Igarashi and H. Nagira. Union types for object-oriented programming. *JOT (www.jot.fm)*, 6(2):47–68, 2007. Special Issue OOPS Track at SAC 2006.
19. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
20. C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall, 2001.
21. 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.
22. L. Liquori and A Spiwack. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, page To appear, 2007.
23. D. Maleyery and J. Aldrich. Combining structural subtyping and external dispatch. In *Electronic proceedings of FOOL/WOOD 2007*, 2007.
24. 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.
25. O. Nierstrasz, S. Ducasse, and N. Schärli. Flattening traits. *JOT (www.jot.fm)*, 5(4):129–148, 2006.
26. M. Odersky. The Scala Language Specification, version 2.4. Technical report, Programming Methods Laboratory, EPFL, Switzerland, 2007.
27. J. Reppy and A. Turon. A foundation for trait-based metaprogramming. In *Electronic proceedings of FOOL/WOOD 2006*, 2006.
28. J. Reppy and A. Turon. Metaprogramming with traits. In *ECOOP 2007*, volume To appear of *LNCS*. Springer, 2007.
29. 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.
30. Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. *SIGPLAN*, 21(11):38–45, 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.