Featherweight Jigsaw
A minimal core calculus
for modular composition of classes*

Giovanni Lagorio, Marco Servetto, and Elena Zucca

DISI, Univ. of Genova, v. Dodecaneso 35, 16146 Genova, Italy
email: {lagorio,servetto,zucca}@disi.unige.it

Abstract. We present FJig, a simple calculus where basic building blocks are classes in the style of Featherweight Java, declaring fields, methods and one constructor. However, inheritance has been generalized to the much more flexible notion originally proposed in Bracha's Jigsaw framework. That is, classes play also the role of modules, that can be composed by a rich set of operators, all of which can be expressed by a minimal core.
We keep the nominal approach of Java-like languages, that is, types are class names. However, a class is not necessarily a structural subtype of any class used in its defining expression.
The calculus allows the encoding of a large variety of different mechanisms for software composition in class-based languages, including standard inheritance, mixin classes, traits and hiding. Hence, FJig can be used as a unifying framework for analyzing existing mechanisms and proposing new extensions.
We provide two different semantics of an FJig program: *flattening* and *direct* semantics. The difference is analogous to that between two intuitive models to understand inheritance: the former where inherited methods are copied into heir classes, and the latter where member lookup is performed by ascending the inheritance chain. Here we address equivalence of these two views for a more sophisticated composition mechanism.

Introduction

Jigsaw is a framework for modular composition largely independent of the underlying language, designed by Gilad Bracha in his seminal thesis [8], and then formalized by a minimal set of operators in module calculi such as [21,3]. In this paper, we define an instantiation of Jigsaw, called Featherweight Jigsaw (FJig for short), where basic building blocks are classes in the style of Java-like languages. That is, classes are collections of fields, methods and constructors, that can be instantiated to create objects; also, class names are used as types (nominal typing).

* This work has been partially supported by MIUR EOS DUE - Extensible Object Systems for Dynamic and Unpredictable Environments.

The motivation for this work is that, even though Jigsaw has been proposed a long time ago and since then it has been greatly influential[1], its design has been never fully exploited in the context of Java-like languages, as recently pointed out as an open question in [4]. Here, we provide a foundational answer to this question, by defining a core language which, however, embodies the key features of Java-like languages, in the same spirit of Featherweight Java [15] (FJ for short). Indeed, formally, a basic class of FJIG looks very much as a class in FJ. However, standard inheritance has been replaced by the much more flexible (module) composition, that is, by the rich set of operators of the Jigsaw framework.

Instantiating Jigsaw on Java-like languages poses some non trivial design problems. Just to mention one (others are discussed in Sect. 1), we keep the nominal approach of Java-like languages, that is, types are class names. However, a class is not necessarily a structural subtype of any class used in its defining expression. While this allows a more flexible reuse, it may prevent the (generalized) inheritance relation to be a subtyping relation. So, the required subtyping relations among classes are declared by the programmer and checked by the type system.

Another challenging issue is the generalization to FJIG of two intuitive models to understand inheritance: one where inherited methods are copied into heir classes, and the other one where member lookup is performed by ascending the inheritance chain. We address the equivalence of these two views for a much more sophisticated composition mechanism. Formally, we provide two different semantics for an FJIG program: *flattening* semantics, that is, by translation into a program where all composition operators have been performed, and *direct* semantics, that is, by formalizing a dynamic look-up procedure.

The paper is organized as follows. Sect. 1 provides an informal introduction to FJIG by using a sugared surface syntax. Sect. 2 introduces a lower level syntax and defines flattening semantics. Sect. 3 defines the type system and states its soundness. Sect. 4 defines direct semantics of FJIG and states the equivalence between the two semantics. In the Conclusion, we summarize the contribution of the paper and briefly discuss related and further work.

A preliminary version of this paper, focused on the equivalence between flattening and direct semantics, and not including the type system, is [17]. An extended version including proofs of results is [16].

1 An informal introduction

In this section we illustrate the main features of FJIG by using a sugared surface syntax, given in Fig. 1. We assume infinite sets of *class names* C, *(member) names* N, and *variables* x. We use the bar notation for sequences, e.g., $\overline{\mu}$ is a metavariable for sequences $\mu_1 \ldots \mu_n$.

[1] Just to mention two different research areas, Jigsaw principles are present in work on extending the ML module system with mutually recursive modules [9,13,14], and Jigsaw operators already included those later used in mixin classes and traits [11,1,20,10,19].

p	$::=$	$\overline{cd}\ \overline{leq}$	program
cd	$::=$	$cmod$ `class` $C\ CE$	class declaration
leq	$::=$	$C <= C'$	subtype declaration
$cmod$	$::=$	`abstract` $\mid \epsilon$	class modifier
CE	$::=$		class expression
		B	basic class
		$\mid C$	class name
		$\mid$ `merge` CE_1, CE_2	merge
		$\mid CE_1$ `override` CE_2	override
		$\mid$ `rename` N `to` N' `in` CE	rename
		$\mid$ `restrict` N `in` CE	restrict
		$\mid$ `hide` N `in` CE	hide
		$\mid \ldots$	
		$\mid CE[\tau]$	`ThisType` wrapper
		$\mid CE[kh\{$`super`$(\overline{e})\}]$	constructor wrapper
N	$::=$	$F \mid M$	member name
kh	$::=$	`constructor`$(\overline{C\ x})$	constructor header
B	$::=$	$\{\tau\ \overline{\varphi}\ \kappa\ \overline{\mu}\}$	basic class
τ	$::=$	`ThisType` $<= C$	`ThisType` constraint
φ	$::=$	$mod\ C\ F;$	field
κ	$::=$	$kh\{\overline{F=e}\}$	constructor
μ	$::=$	$mod\ C\ M\ (\overline{C\ x})$`{return` $e;$`}`	
		$\mid$ `abstract` $C\ M(\overline{C\ x});$	method
mod	$::=$	`abstract` $\mid$ `virtual` $\mid$ `frozen` $\mid$ `local`	member modifier
e	$::=$		expression
		x	variable
		$\mid e.F$	client field access
		$\mid e.M(\overline{e})$	client method invocation
		$\mid F$	internal field access
		$\mid M(\overline{e})$	internal method invocation
		$\mid$ `new` $C(\overline{e})$	object creation

Fig. 1. FJIG (surface) syntax

This syntax is designed to keep a Java-like flavour as much as possible. In the next section we will use a lower-level representation, which allows to formalize the semantics in a simpler and natural way.

We will first revise Jigsaw features in the context of FJIG, then discuss some issues that are specific to the instantiation on Java-like languages.

Basic classes Jigsaw is a programming paradigm based on (module) composition, where a basic module (in our case, a class) is a collection of components (in our case, members), which can be of four different kinds, indicated by a modifier: `abstract`, `virtual`, `frozen`, and `local`. A method has no body if and only if its modifier is `abstract`. The meaning of modifiers is as follows:

– An `abstract` member has no definition, and is expected to be defined later when composing the class with others.
– A `virtual` or `frozen` member has a definition, which can be changed by using the composition operators. However, the redefinition of a `frozen` member does not affect the other members, which still refer to its original definition.
– Finally, as the name suggests, a `local` member cannot be selected by a client[2], and is not affected by composition operators, hence its definition cannot be changed.

We assume by default the modifier `frozen` for fields and `virtual` for methods. A class having at least one `abstract` member must be declared `abstract`. The following example shows two abstract basic classes.[3]

```
abstract class A {
  abstract int M1();
  int M2() { return M1() + M3(); }
  local int M3() { return 1; }
}
abstract class B {
  abstract int M2();
  frozen int M1() { return 1 + M2(); }
}
```

Merge and override operators A concrete class can be obtained by applying the `merge` operator as follows:

```
class C merge A, B
```

This declaration is equivalent to the following:

```
class C {
  frozen int M1() { return 1 + M2(); }
  int M2() { return M1() + M3(); }
  local int M3() { return 1; }
}
```

Conflicting definitions for the same non-local member are not permitted, whereas `abstract` members with the same name are shared. Members can be selected by client code unless they are `local`, that is, we can write, e.g., `new C().M2()` but not `new C().M3()`. To show the difference between `virtual` and `frozen` members, in the next examples we use the `override` operator, a variant of `merge` where conflicts are allowed and the left argument has the precedence.

```
class D1
  { int M2() { return 2; } } override C
```

An invocation `new D1().M2()` will evaluate to `2`, and an invocation `new D1().M1()` to `3`. On the other hand, in this case:

[2] Note the difference with `private` modifier in Java, which allows client selection when clients are of the same class, see more details in the sequel.

[3] To write more readable examples, we assume that the primitive type `int` and its operations are available.

```
class D2
  { int M1() { return 3; } } override C
```

an invocation `new D2().M1()` will evaluate to `3`, *but* an invocation `new D2().M2()` will not terminate, since the internal invocation `M1()` in the body of `M2()` still refers to the old definition.

Client and internal member selection In a programming paradigm based on module composition, a module component can be either selected by a client, or used by other components inside the module itself. Correspondingly, in FJIG we distinguish between *client* field accesses and method invocations, which specify a receiver, and *internal* field accesses and method invocations, whose implicit receiver is the current object. Note that $e.M(\ldots)$ behaves differently from $M(\ldots)$ even in the case e denotes an object of the same class (that is, internal selection *does not* correspond to selection of `private` members as in, e.g., Java). For instance, consider the following class, where we use the operator `rename`, which changes the name of a member.

```
class E merge
  (rename M1 to M4 in {
    int M1() { return 1; }
    int M2() { return M1(); }
    int M3() { return new E().M1(); }
}), { int M1() { return 3; } }
```

An invocation `new E().M2()` returns `1`, since the internal invocation in the body of `M2` refers to the method now called `M4`. However, an invocation `new E().M3()` returns `3`, since the client invocation in the body of `M3` refers to method `M1` in `E`. Note that this does not even coincide with privateness on a "per object" basis as, e.g., in Smalltalk, since this would be the case even with a client invocation `e.M1()`, where e denotes, as special case, the current object.
Other operators of the Jigsaw framework, besides the ones mentioned above, are `restrict`, which eliminates the definition for a member[4], and `hide`, which makes a member no longer selectable from the outside. We refer to [8] and [3] for more details. All these operators and many others can be easily encoded (see [3]) by using a minimal set of *primitive* operators: *sum*, *reduct*, and *freeze*, which will be formally defined in next section.
We discuss now the issues specific to the instantiation on Java-like classes.

Fields and constructors It turns out that the above modifiers can be smoothly applied to fields as well, with analogous meaning, as shown by the following example which also illustrates how constructors work.

```
class A1 {
  abstract int F1; virtual int F2; int F3;
  constructor(int x) { F2 = x; F3 = x; }
  int M() { return F2 + F3; }
```

[4] Indeed, CE_1 `override` $CE_2 =$ `merge` CE_1, `restrict` N_1 `in` $\ldots$ `restrict` N_k `in` CE_2 where $N_1, \ldots N_k$ are the common members.

```
}
class C1 {
  int F1; int F2; int F3;
  constructor(int x) { F1 = x+1; F2 = x+1; F3 = x+1; }
} override A1
```

A basic class defines one[5] constructor which specifies a sequence of parameters and a sequence of initialization expressions, one for each non-abstract field. We assume a default constructor with no parameters for classes having no fields. Note the difference with FJ, where the class constructor has a canonical form (parameters exactly correspond to fields). This would be inadequate in our framework since object layout must be hidden to clients. In order to be composed by merge/overriding, two classes should provide a constructor with the same parameter list (if it is not the case, a *constructor wrapper* can be inserted, see the last example of this section), and the effect is that the resulting class provides a constructor with the same parameter list, that executes both the original constructors. An instance of class `C1` has five fields (`A1.F2`, `A1.F3`, `C1.F1`, `C1.F2`, `C1.F3`), and an invocation `new C1(5).M()` will return `11`, since `F2` in the body of `M` refers to the field declared in `C1` (initialized with `5+1`), while `F3` refers to the field declared in `A1` (initialized with `5`).[6]
Classes composed by merge/overriding can share the same field, provided it is abstract in all except (at most) one. Note that this corresponds to *sharing* fields as in, e.g., [5]; however, in our framework we do not need an ad-hoc notion.

Inheritance and subtyping Since our aim is to instantiate the Jigsaw framework on a Java-like language, we keep a nominal approach, that is, types are class names. However, subtyping *does not* coincide with the generalized inheritance relation, since some of the composition operators (e.g., renaming) do not preserve structural subtyping. Hence, we assume that a program includes a sequence of subtyping relations among classes explicitly declared by the programmer, and the type system checks, for each $C <= C'$ subtype declaration, that the relation can be safely assumed since C is a structural subtype of C'.

Type of the current object The following code

```
{ C M() { return this; } }
```

can be safely inherited only by classes which are a subtype of `C`. To ensure this, basic classes can declare a `ThisType` constraint:

```
{ ThisType <= C;
  C M() { return this; }
}
```

This constraint is used to typecheck the occurrences of `this` inside method bodies. Moreover, the constraint is checked when inheriting the code:

[5] Since overloading is not allowed.

[6] Note that an overriden member, such as `A1.F2`, could still be selected, as usual, by a `super` mechanism, which can be encoded in Jigsaw, notably by renaming [2].

```
class C {
   ThisType <= C;
   C M() {return this;}
}
class D  ... C ... //ok only if D <= C
```

The `ThisType` constraint can be strengthened by the `ThisType` wrapping operator

```
C [ThisType <= D] //ok only if D <= C
```

We assume a default constraint `ThisType` <= `Object`, where `Object` is a predefined class with no members.

To conclude this section, we show a more significant example, where we also assume to have the type `void` and some statements in the syntax.
The following class `DBSerializable`, an example of the pattern *template method* [12], contains the method `saveToDB`, which writes the object serialized representation onto a database. While the behaviour of `saveToDB` is fixed, the details on how to open the connection are left unspecified, and the implementation of the method `serialize` can be changed.[7] This is reflected by the method modifiers. Class `DBConnection` is a given library class.

```
abstract class DBSerializable {
  abstract DBConnection openConnection();
  virtual void serialize(DBConnection c) {}
  frozen void saveToDB() {
    DBConnection connection = openConnection();
    // ...
    serialize(connection);
    connection.close();
  }
}
```

Suppose we want to specialize the class `DBSerializable` for the DB server MySQL. We can create this specialization, called `MySQLSerializable`, in two steps: first, we provide an implementation of method `openConnection` with the specific code for MySQL, then we *hide* it, since clients of `MySQLSerializable` should never invoke this method directly. We start by defining an auxiliary class `_MySQLSerializable`, merging `DBSerializable` with an anonymous basic class:

```
class _MySQLSerializable
  merge
    DBSerializable[ constructor(String cs) {
                      super()
                  } ],
    { local String connectionString;
      constructor(String cs) { connectionString = cs; }
      virtual DBConnection openConnection() {
        /* ... use connectionString ... */}
    }
```

[7] This method could be declared abstract as well.

Note the use of the constructor wrapper: the constructor of the anonymous basic class has a `String` parameter, whereas that of the class `DBSerializable`, which has no fields, is the default (parameterless) constructor. Hence, a constructor wrapper is inserted, so that the classes we are merging have both a constructor with the same parameters. This allows to create objects of the new class with expressions like `new _MySQLSerializable("someConnectionString...")`. As mentioned before, the class `_MySQLSerializable` provides, along the method `saveToDB`, the method `openConnection` that we can hide as follows:

```
class MySQLSerializable
  hide openConnection in _MySQLSerializable
```

Consider now the following class `Person`, providing a method, named `write`, to serialize its objects to a database:

```
class Person {  // ...
  frozen void write(DBConnection c) {
     /* serializes the data on c*/}
}
```

Notwithstanding the inherited method `DBSerializable.saveToDB` writes the data by invoking the method `serialize` and not `write`, using the class `Person` with `MySQLSerializable` is not a problem, since we can rename the method before merging the two classes:

```
class MySQLSerializablePerson
  hide serialize in
    (rename write to serialize in Person)
      [constructor(String cs){super()}]
    override MySQLSerializable
```

2 FJIG calculus

In this section we formally define the (flattening) semantics of FJIG. To this aim, we use a different representation for basic classes w.r.t. the surface syntax given in Fig. 1. That is, instead of having explicit modifiers, their semantics is encoded by distinguishing between *external* and *internal* member names. Internal names are used to refer to members inside code (method bodies), whereas external names are used in class composition via operators and in selection of members by clients. Correspondingly, basic classes include, besides previous components which are collected in the *local part*, an *input map* from internal to external names, and an *output map* from external to internal names. Intuitively, the input map translates required internal names to external names which are actually required from other classes, and the output map translates provided external names to internal names which actually provide their definitions. We could have alternatively expressed the semantics directly on the surface language, getting a more FJ-like flavour. However, the representation with i/o maps has some advantages: a clean distinction between internal names, which can be α-renamed, and external names, as in the tradition of module calculi [21,3]; renaming (reduct operator)

can be modeled without changing (occurrences of names in) code; in general, operators can be modeled in a uniform way, whereas the other representation would require a case analysis on the four kinds of members.

The syntax of the calculus is given in Fig. 2. Besides class names, (external) names and variables, we assume an infinite set of *internal (member) names* n. A program consists of two components: a sequence of *class declarations* (class name and class expression), as in FJ, and a sequence of *subtype declarations*. We assume that no class is declared twice and order is immaterial, hence we can write $p(C)$ for the class expression associated with C. Class expressions CE are basic classes B, class names C, or are inductively constructed by a set of composition operators. Let us say that C "inherits from" C' if the class expression associated with C contains a subterm C', or, transitively, C'' which inherits from C'. In a well-formed program, we require this generalized inheritance relation to be acyclic, exactly as for standard inheritance. Input and output maps are represented as sequences of pairs where the first element has a type annotation. In an input map, internal names which are mapped to the same external name are required to have the same annotation, whereas this is not required in output names, that is, the same member can be exported under different names with different types, see the type system in next section. Renamings σ are maps from (annotated) external names into (annotated) external names, represented as sequences of pairs; pairs of form $\_ \mapsto N{:}T$ are used to represent non-surjective maps. We use some shorter keywords w.r.t. the surface syntax, and expressions include *runtime expressions*, that is, (pre-)objects and blocks.

We denote by *dom* and *img* the domain and image of a map, respectively. Given a basic class $[\iota \,|\, o \,|\, \rho]$, with $\rho = \{\tau\ \overline{\varphi}\ \kappa\ \overline{\mu}\}$, we denote by $dom(\overline{\mu})$ and $dom(\overline{\varphi})$ the sets of internal names declared in $\overline{\mu}$ and $\overline{\varphi}$, respectively, which are assumed to be disjoint. The union of these two sets, denoted by $dom(\rho)$, is the set of *local* names. An internal name n is, instead, *abstract* if $n{\in}dom(\iota), \iota(n){\notin}dom(o)$, and *virtual* if $\iota(n){\in}dom(o)$. An external name N is *abstract* if $N{\in}img(\iota){\setminus}dom(o)$, *virtual* if $N{\in}img(\iota){\cap}dom(o)$, *frozen* if $N{\in}dom(o){\setminus}img(\iota)$. In a well-formed basic class, local names must be distinct from abstract/virtual internal names, that is, $dom(\iota){\cap}dom(\rho){=}\emptyset$. Moreover, $img(o){\subseteq}dom(\rho)$, and, denoting by $names(e)$ the set of internal names in an expression e, $names(e){\subseteq}dom(\iota){\cup}dom(\rho)$ for each method body e.

A basic class of the surface language can be easily encoded in the calculus as follows. For each member name N we assume (at most) a corresponding external name N and (at most) two internal names n, n', depending on the member kind, as detailed below. Client references to N are unaffected, whereas internal references are translated according to the member kind:

- if N is abstract, then there is an association $n{\mapsto}N$ in the input map, and internal references are translated by n,
- if N is virtual, then there is an association $n{\mapsto}N$ in the input map, an association $N{\mapsto}n'$ in the output map, a definition for n' in ρ, and internal references are translated by n,

p	$::=$	$\overline{cd}\ \overline{leq}$	program
cd	$::=$	$C \mapsto CE$	class declaration
leq	$::=$	$C \leq C'$	subtype declaration
CE	$::=$	$B \mid C \mid$	class expression
		$CE_1 + CE_2$	sum
		$\mid {}_{\sigma^\iota\mid}CE_{\mid\sigma^o}$	reduct
		$\mid freeze_N CE$	freeze
		$\mid CE[\mathtt{K}(\overline{C\ x})\{\overline{e}\}] \mid CE[\mathtt{TT}{\leq}C]$	constructor and `ThisType` wrappers
σ	$::=$	$\overline{N{:}T{\mapsto}N'{:}T'}, \overline{\_ \mapsto N{:}T}$	renaming
N	$::=$	$F \mid M$	external member name
T	$::=$	$C \mid MT$	member type
MT	$::=$	$\overline{C} \to C$	method type
B	$::=$	$[\iota \mid o \mid \rho]$	basic class
ι	$::=$	$\overline{n{:}T{\mapsto}N}$	input map
o	$::=$	$\overline{N{:}T{\mapsto}n}$	output map
n	$::=$	$f \mid m$	internal member name
ρ	$::=$	$\{\tau\ \overline{\varphi}\ \kappa\ \overline{\mu}\}$	local part
τ	$::=$	$\mathtt{TT}{\leq}C$	`ThisType` constraint
φ	$::=$	$C\ f;$	field
κ	$::=$	$\mathtt{K}(\overline{C\ x})\{\overline{f{=}e}\}$	constructor
μ	$::=$	$C\ m(\overline{C\ x})\{\mathtt{return}\ e;\}$	method
e	$::=$	$x \mid e.F \mid e.M(\overline{e}) \mid f \mid m(\overline{e}) \mid \mathtt{new}\ C(\overline{e})$	expression
		$\mid [\overline{\mu}; v \mid e]$	block
		$\mid C(\overline{f{=}e})$	(pre-)object
v, v^C	$::=$	$C(\overline{f{=}e})$	value (object)

Fig. 2. Syntax

- if N is frozen, then there is an association $N{\mapsto}n'$ in the output map, a definition for n' in ρ, and internal references are translated by n'.
- if N is local, then there is a definition for n' in ρ, and internal references are translated by n'.

Inside constructor bodies, a field name F on the left-hand side is always translated by f' (and internal member selection is forbidden).
For instance, the class `C` shown in the previous section is translated by

$[m_2{:}() \to \mathtt{int} \mapsto M_2 \mid M_1{:}() \to \mathtt{int} \mapsto m_1', M_2{:}() \to \mathtt{int} \mapsto m_2', \mid \rho]$
$\rho = \{$
  $\mathtt{TT}{\leq}\mathtt{Object}\quad \mathtt{K}()\{\}$
  $\mathtt{int}\ m_1'()\{\mathtt{return}\ 1 + m_2();\}$
  $\mathtt{int}\ m_2'()\{\mathtt{return}\ m_1'() + m_3'();\}$
  $\mathtt{int}\ m_3'()\{\mathtt{return}\ 1;\}$
$\}$

We describe now the two kinds of runtime expressions introduced in the calculus. Expressions of form $C(\overline{f{=}e})$ denote a *pre-object* of class C, where for each field f there is a corresponding initialization expression. Note the difference with the form `new` $C(\overline{e})$, which denotes a constructor invocation, whereas in FJ objects can be identified with object creation expressions where arguments are values. As already noted, in FJ it is possible, and convenient, to take this simple and nice solution, since the structure of the instances of a class is globally visible to the whole program. In FJIG, instead, object layout must be hidden to clients, hence constructor parameters have no a priori relation with fields.
Values of the calculus are *objects*, that is, pre-objects where all initialization expressions are (in turn) values. We use both v^C and v as metavariables for values of class C, the latter when the class is not relevant.
Moreover, runtime expressions also include *block* expressions of the form $[\overline{\mu}; v \mid e]$, which model the execution of e where method internal names are bound in $\overline{\mu}$ and field internal names in the current object v. Hence, denoting by $dom(v)$ the set $\{f_1, \ldots, f_n\}$ if $v{=}C(f_1{=}v_1 \ldots f_n{=}e_n)$, a block expression is well-formed only if $names(e){\subseteq}dom(\overline{\mu}){\cup}dom(v)$ (hence $names([\overline{\mu}; v \mid e]) = \emptyset$) and these two sets are disjoint.

The semantics of an expression e in the context of a program p can be defined in two different ways. The former, which we call *flattening semantics* and illustrate in this section, is given in two steps. First, p is reduced to a *flat* program p', that is, a program where every class is basic. To this end, operators are performed and the occurrences of class names are replaced by their defining expressions. Then, e is reduced in the context of p'. Note that in this case dynamic look-up is always trivial, that is, a class member (e.g., a method) can be always found in the class of the receiver. In next section, we define an alternative *direct* semantics, where expressions are reduced in the context of non flat programs, hence where dynamic look-up is non trivial.
Flattening rules are defined in the top section of Fig. 3. We omit subtype declarations for simplicity since they do not affect semantics.
The first two rules define reduction steps of programs, which can be obtained either by reducing one of the class expressions, or, if some class C has already been reduced to a basic class B, by replacing by B all occurrences of C as class expression.
The remaining rules define reduction steps of class expressions. Rules for sum, reduct and freeze operators are essentially those given in [3], to which we refer for more details. We omit standard contextual closure for brevity.
The expression o_1, o_2 is well-formed only if the two maps have disjoint domains, and analogously for other maps. Hence, rule (SUM) can only be applied (implicit side conditions) when the two sets of internal names are disjoint, as are the sets of output names. The former condition can be always satisfied by an appropriate α-conversion, whereas the latter corresponds to a conflict that the programmer can only solve by an explicitly renaming (reduct operator). Note that `ThisType` constraints and constructor parameters are required to be the same, in order

(CDEC1) $$\frac{CE \longrightarrow CE'}{(p, C \mapsto CE) \longrightarrow (p, C \mapsto CE')}$$

(CDEC2) $$\frac{}{(p, C \mapsto B) \longrightarrow (p[B/C], C \mapsto B)}$$

(SUM) $$\frac{}{[\iota_1 \,|\, o_1 \,|\, \rho_1] + [\iota_2 \,|\, o_2 \,|\, \rho_2] \longrightarrow [\iota_1, \iota_2 \,|\, o_1, o_2 \,|\, \rho]}$$ $\rho_i = \{\tau\ \overline{\varphi}_i\ \mathtt{K}(\overline{C\ x})\{(\overline{f{=}e})_i\}\ \overline{\mu}_i\}$, $i \in \{1,2\}$; $\rho = \{\tau\ \overline{\varphi}_1, \overline{\varphi}_2\ \mathtt{K}(\overline{C\ x})\{(\overline{f{=}e})_1, (\overline{f{=}e})_2\}\ \overline{\mu}_1, \overline{\mu}_2\}$

(REDUCT) $$\frac{}{{}_{\sigma^\iota}[\![\iota \,|\, o \,|\, \rho]\!]_{|\sigma^o} \longrightarrow [\sigma^\iota \circ \iota \,|\, o \circ \sigma^o \,|\, \rho]}$$

(FREEZE) $$\frac{}{\begin{array}{l} \mathit{freeze}_N[\iota, n_1{:}T{\mapsto}N \ldots n_k{:}T{\mapsto}N \,|\, o \,|\, \rho] \longrightarrow \\ \quad [\iota \,|\, o \,|\, \rho[n'/n_1] \ldots [n'/n_k]] \end{array}}$$ $n' = o(N)$, $N \notin \mathit{img}(\iota)$

(TT WRAP) $$\frac{}{[\iota \,|\, o \,|\, \{\mathtt{TT}{\leq}C'\ \overline{\varphi}\ \kappa\ \overline{\mu}\}][\mathtt{TT}{\leq}C] \longrightarrow [\iota \,|\, o \,|\, \{\mathtt{TT}{\leq}C\ \overline{\varphi}\ \kappa\ \overline{\mu}\}]}$$

(K WRAP) $$\frac{}{[\iota \,|\, o \,|\, \rho][\mathtt{K}(\overline{\mathtt{D}\ \mathtt{y}})\{\overline{e}\}] \longrightarrow [\iota \,|\, o \,|\, \rho']}$$ $\overline{x} = x_1 \ldots x_n$; $\rho = \{\tau\ \overline{\varphi}\ \mathtt{K}(C_1\ x_1 \ldots C_n\ x_n)\{\overline{f{=}e}\}\ \overline{\mu}\}$; $\rho' = \{\tau\ \overline{\varphi}\ \mathtt{K}(\overline{\mathtt{D}\ \mathtt{y}})\{\overline{f{=}e[\overline{e}/\overline{x}]}\}\ \overline{\mu}\}$

(CTX) $$\frac{e \longrightarrow_p e'}{\mathcal{E}\{e\} \longrightarrow_p \mathcal{E}\{e'\}}$$ (CLIENT-FIELD) $$\frac{}{v^C.F \longrightarrow_p [\overline{\mu}; v^C \,|\, f]}$$ $p(C) = [\iota \,|\, o \,|\, \{\tau\ \overline{\varphi}\ \kappa\ \overline{\mu}\}]$, $o(F) = f$

(CLIENT-INVK) $$\frac{}{v^C.M(\overline{v}) \longrightarrow_p [\overline{\mu}; v^C \,|\, m(\overline{v})]}$$ $p(C) = [\iota \,|\, o \,|\, \{\tau\ \overline{\varphi}\ \kappa\ \overline{\mu}\}]$, $o(M) = m$

(INT-FIELD) $$\frac{}{[\overline{\mu}; v \,|\, \mathcal{E}\{f\}] \longrightarrow_p [\overline{\mu}; v \,|\, \mathcal{E}\{v_i\}]}$$ $f \notin \mathit{HB}(\mathcal{E})$, $v = C(f_1{=}v_1 \ldots f_n{=}v_n)$, $f{=}f_i$

(INT-INVK) $$\frac{}{[\overline{\mu}; v^C \,|\, \mathcal{E}\{m(\overline{v})\}] \longrightarrow_p [\overline{\mu}; v^C \,|\, \mathcal{E}\{e[\overline{v}/\overline{x}][v^C/\mathtt{this}]\}]}$$ $m \notin \mathit{HB}(\mathcal{E})$, $\overline{\mu}(m) = \langle \overline{x}, e \rangle$

(OBJ-CREATION) $$\frac{}{\mathtt{new}\ C(\overline{v}) \longrightarrow_p C(\overline{f{=}e[\overline{v}/\overline{x}]})}$$ $p(C) = [\emptyset \,|\, o \,|\, \rho]$, $\rho = \{\tau\ \overline{\varphi}\ \mathtt{K}(C_1\ x_1 \ldots C_n\ x_n)\{\overline{f{=}e}\}\ \overline{\mu}\}$, $\overline{x} = x_1 \ldots x_n$

(EXIT-BLOCK) $$\frac{}{[\overline{\mu}; v \,|\, e] \longrightarrow_p e}$$ $\mathit{names}(e) = \emptyset$

Fig. 3. Flattening semantics

both to get a commutative operator and to keep the calculus minimal; indeed, this can be always achieved by using wrapping operators.

In rule (REDUCT), new input and output names are chosen, modeled by $img(\sigma^\iota)$ and $dom(\sigma^o)$, respectively. Old input names are mapped in new input names by σ^ι, whereas new output names are mapped into old output names by σ^o. Input names can be shared or added, whereas output names can be duplicated or removed. The symbol $\circ$ denotes composition of maps, which is well-formed only if type annotations are the same and the annotation of the new name is kept in the resulting map.

In rule (FREEZE), association from internal names into N are removed from the input map, and occurrences of these names in method bodies are replaced by the local name of the corresponding definition, thus eliminating any dependency on N. The second side condition ensures that we actually take *all* such names.

Rules for constructor and `ThisType` wrapping just correspond to changing the constructor and the `ThisType` constraint for a class, respectively. In (K WRAP), n is the arity of the old constructor, and the body of the new constructor has n initialization expressions, as implicitly imposed by the well-formedness of multiple substitution $\overline{e}$ for $\overline{x}$. We chose a permissive semantics for `ThisType` wrapping, alternatively we could perform a runtime check on the relation between C and C'.

Reduction rules are given in the second section of Fig. 3.

The first rule is the standard contextual closure, where $\mathcal{E}$ denotes a one-hole context and $\mathcal{E}\{e\}$ denotes the expression obtained by filling the hole by e.

Client field accesses and method invocations are reduced in two steps. First, they are reduced to a block where the current object is the receiver and the expression to be executed is the corresponding internal member selection on the name found in the receiver's class; moreover, methods found in the receiver's class are copied into the block and used for resolving further internal method invocations.[8] Then, the following two rules can be applied.

An internal field access can only be reduced if it appears inside a block. In this case, it is replaced by the corresponding field of the current object. The first side condition says that the occurrence of f or m in the position denoted by the hole of the context $\mathcal{E}$ is free (that is, not captured by any binder around the hole), hence ensures that it is correctly bound to the current object in the first enclosing block. The standard formal definition of HB is omitted. For instance, in the expression $[\overline{\mu}; v \,|\, m(f, [\overline{\mu}'; v' \,|\, f])]$, the first occurrence of f denotes a field of the object v, whereas the second occurrence denotes a field of the object v'. Analogously, an internal method invocation is replaced by the corresponding body, found in $\overline{\mu}$, where parameters are replaced by arguments and `this` by the current object. We denote by $\overline{\mu}(m)$ the pair $\langle x_1 \ldots x_n, e\rangle$ if $\overline{\mu}$ contains a (unique) method $C\ m(C_1\ x_1\ \ldots C_n\ x_n)\{\texttt{return}\ e;\}$.

[8] Alternatively, the method body corresponding to an internal name could be again found in the basic class of the receiver; we choose this model because it can be better generalized to direct semantics, see the following.

Note that there are two kinds of references to the current object in a method body: through the keyword `this` (in client member selection, or in a non-receiver position, e.g. `return this`), and through internal names. Whereas the former can be substituted at invocation time, as in FJ, the latter are modeled by a block, otherwise we would not be able to distinguish, among the objects of form v^C, those which actually refer to the original receiver of the invocation.
In rule (OBJ-CREATION), note that only classes where all members are frozen can be instantiated. This is a simplification: the execution model could be easily generalized to handle internal member selection on a virtual internal name by retrieving the input map as well in blocks (in rules (CLIENT-FIELD) and (CLIENT-INVK)) and adding two reduction rules which, roughly, reduce such an internal field access/method invocation into the corresponding client member selection. We preferred to stick to an equivalent simpler model which, assuming that all classes have been frozen before being instantiated, avoids these redundant lookup steps.
Finally, in (EXIT-BLOCK), a block can be eliminated when the enclosed expression does no longer contain internal member selections, hence in particular when a value is obtained.
Examples illustrating flattening semantics (in comparison with direct semantics) will be provided in Sect. 4.

3 Type system

The type system uses four kinds of type environments, shown in Fig. 4.

Δ	$::= \overline{C{:}CT}\ \overline{leq}$	class type environment
CT	$::= [\Sigma^\iota; \Sigma^o; \overline{C}; C]$	class type
Γ	$::= \overline{n{:}T}$	internal type environment
Π	$::= \overline{x{:}C}$	parameter type environment
Σ	$::= \overline{N{:}T}$	signature
Δ^r	$::= \overline{C{:}\Gamma}$	runtime class type environment

Fig. 4. Type environments

A class type environment is a pair consisting of a map from class names into class types and a sequence of subtype declarations. A class type is a 4-tuple consisting of input and output signatures, constructor type and type of `this`. We use the abbreviated notations $C \leq C' \in \Delta$ and $\Delta(C) = CT$.
Signatures are maps from external names into types.
We denote by $mtype(\Delta, C, N)$ the type of member named N in $\Delta(C)$, which is the output type[9] for a defined member, the input type for an abstract member.

[9] To provide a richer interface to clients.

Internal type environments map internal names to types. Parameter type environments map variables (parameters) into class names. Finally, runtime class type environments map class names to internal type environments.

Typing rules in Fig. 5 define judgments $\vdash p{:}\Delta$ for programs and $\Delta \vdash CE{:}CT$ for class expressions.

In (PROG-T), a program has type Δ if each declared class C has type $\Delta(C)$ w.r.t. Δ, `ThisType` constraints are satisfied, and declared subtyping relations are safe. The judgment $\Delta \vdash C \leq C'$ checks whether C and C' are in the reflexive and transitive closure of subtyping declarations in Δ. The judgment $\Delta \vdash C \leq C'$ OK checks whether C is a structural subtype of C. The straightforward definition of these judgments is omitted (see [16]).

In (BASIC-T), we denote by Σ^ι and Σ^o the signatures extracted from ι and o, respectively; analogously, we denote by Γ^ι,$\Gamma^{\overline{\mu}}$ and $\Gamma^{\overline{\varphi}}$ the internal type environments extracted from ι, $\overline{\mu}$ and $\overline{\varphi}$, respectively.

A basic class is well-typed w.r.t. Δ under three conditions. First, methods have their declared types w.r.t. Δ, the internal type environment assigning to member internal names their annotations, and the type in the `ThisType` constraint (assumed as type for `this`). Second, the constructor has its declared type w.r.t. Δ and the internal type environment assigning to internal field names their annotations. Finally, type annotations in input signature, output signature and local part must be consistent, that is, a virtual member can be used inside the class with a supertype of its exported type (first side condition), and a member can be exported with a supertype of its internal type (second side condition).

Typing rules for sum, reduct and freeze are based on those in [3]. Rule (SUM-T) imposes the same constructor type and `ThisType` constraint, and disjoint output signatures. In (REDUCT-T), the judgment $\Delta \vdash \sigma{:}\Sigma \to \Sigma'$ means that, if σ maps $N{:}T$ into $N'{:}T'$, then $\Delta \vdash T' \leq T$ holds. Hence, the side condition allows a member to be imported with a more specific type, and exported with a more general type. Analogously, rule (THIS-TYPE-T) allows the type of `this` to become more specific.

Typing rules in Fig. 6 define the judgment $\Delta; \Gamma; \Pi \vdash e{:}C$ for well-typed expressions. They are analogous to FJ rules. However, note that member type is found in receiver's class for client member selection, whereas it is found in the internal type environment for internal member selection. Also, note that (NEW-T) requires a class to have an empty input signature in order to be instantiated (see comment to rule (OBJ-CREATION) in previous section).

Finally, typing rules in Fig. 7 define the judgment $\Delta; \Delta^r; \Gamma; \Pi \vdash e{:}C$ for well-typed runtime expressions. These expressions are typed using an additional type environment Δ^r, which gives for each class the types of its internal field names. Rule (BLOCK-T) checks that the current object is well-typed and, moreover, that the enclosed method declarations and expression are well-typed in the internal type environment corresponding to the current object's class in Δ^r. In this case, the type of the block is that of the enclosed expression. Rule (PRE-OBJ-T) checks that each initialization expressions has a subtype of the type of the corresponding field internal name, found in the internal type environment associated to the

$$\text{(PROG-T)}\ \frac{\begin{array}{l}\Delta \vdash CE_i{:}CT_i \quad \forall i \in 1..n \\ \Delta \vdash C_i \leq C_i^\tau \ \forall i \in 1..n \\ \Delta \vdash C_i' \leq C_i'' \ \text{OK} \ \forall i \in 1..k\end{array}}{\vdash C_1 \mapsto CE_1 \ldots C_n \mapsto CE_n\ \overline{leq}{:}\Delta}\ \begin{array}{l}\overline{leq} = C_1' \leq C_1'' \ldots C_k' \leq C_k'' \\ \Delta = C_1{:}CT_1 \ldots C_n{:}CT_n\ \overline{leq} \\ CT_i = [\_;\_;\_;C_i^\tau]\end{array}$$

$$\text{(CNAME-T)}\ \frac{}{\Delta \vdash C{:}CT}\ \Delta(C) = CT$$

$$\text{(BASIC-T)}\ \frac{\begin{array}{c}\Delta; \Gamma^\iota, \Gamma^{\overline{\mu}}, \Gamma^{\overline{\varphi}}; C \vdash \overline{\mu}{:}\Gamma^{\overline{\mu}} \\ \Delta; \Gamma^{\overline{\varphi}} \vdash \kappa{:}\overline{C}\end{array}}{\Delta \vdash [\iota \,|\, o \,|\, \{\mathtt{TT}{\leq}C\ \overline{\varphi}\ \kappa\ \overline{\mu}\}]{:}[\Sigma^\iota; \Sigma^o; \overline{C}; C]}\ \begin{array}{l}\Delta \vdash \Sigma^o(N) \leq \Sigma^\iota(N) \\ \qquad \forall N \in img(\iota) \cap dom(o) \\ \Delta \vdash (\Gamma^{\overline{\varphi}}, \Gamma^{\overline{\mu}})(o(N)) \leq \Sigma^o(N) \\ \qquad \forall N \in dom(o)\end{array}$$

$$\text{(METHODS-T)}\ \frac{\Delta; \Gamma; C \vdash \mu_i{:}MT_i\ \forall\ i \in 1..n}{\Delta; \Gamma; C \vdash \overline{\mu}{:}\Gamma^{\overline{\mu}}}\ \begin{array}{l}\overline{\mu} = \mu_1 \ldots \mu_n \\ \Gamma^{\overline{\mu}} = m_1{:}MT_1 \ldots m_n{:}MT_n\end{array}$$

$$\text{(METHOD-T)}\ \frac{\Delta; \Gamma; \mathtt{this}{:}C, x_1{:}C_1 \ldots x_n{:}C_n \vdash e{:}C'}{\begin{array}{l}\Delta; \Gamma; C \vdash\ C_0\ m(C_1\ x_1 \ldots C_n\ x_n)\{\mathtt{return}\ e;\}{:} \\ \qquad C_1 \ldots C_n \to C_0\end{array}}\ \Delta \vdash C' \leq C_0$$

$$\text{(K-T)}\ \frac{\Delta; \emptyset; x_1{:}C_1 \ldots x_n{:}C_n \vdash e_i{:}C_i''\ \forall i \in 1..k}{\Delta; f_1{:}C_1' \ldots f_k{:}C_k' \vdash \kappa{:}C_1 \ldots C_n}\ \begin{array}{l}\kappa = \mathtt{K}(C_1\ x_1 \ldots C_n\ x_n)\{f_1 = e_1 \ldots f_k = e_k\} \\ \Delta \vdash C_i'' \leq C_i'\ \forall i \in 1..k\end{array}$$

$$\text{(SUM-T)}\ \frac{\begin{array}{c}\Delta \vdash CE_1{:}[\Sigma_1^\iota; \Sigma_1^o; \overline{C}; C] \\ \Delta \vdash CE_2{:}[\Sigma_2^\iota; \Sigma_2^o; \overline{C}; C]\end{array}}{\Delta \vdash CE_1 + CE_2{:}[\Sigma_1^\iota, \Sigma_2^\iota; \Sigma_1^o, \Sigma_2^o; \overline{C}; C]}\ dom(\Sigma_1^o) \cap dom(\Sigma_2^o) = \emptyset$$

$$\text{(REDUCT-T)}\ \frac{\Delta \vdash CE{:}[\Sigma_1^\iota; \Sigma_1^o; \overline{C}; C]}{\Delta \vdash {}_{\sigma^\iota|}CE_{|\sigma^o}{:}[\Sigma^\iota; \Sigma^o; \overline{C}; C]}\ \begin{array}{l}\Delta \vdash \sigma^\iota{:}\Sigma_1^\iota \to \Sigma^\iota \\ \Delta \vdash \sigma^o{:}\Sigma^o \to \Sigma_1^o\end{array}$$

$$\text{(FREEZE-T)}\ \frac{\Delta \vdash CE{:}[\Sigma^\iota, N{:}T; \Sigma^o; \overline{C}; C]}{\Delta \vdash \mathit{freeze}_N CE{:}[\Sigma^\iota; \Sigma^o; \overline{C}; C]}\ N{\in}dom(\Sigma^o)$$

$$\text{(TT-WRAP-T)}\ \frac{\Delta \vdash CE{:}[\Sigma^\iota; \Sigma^o; \overline{C}; C']}{\Delta \vdash CE[\mathtt{TT}{\leq}C]{:}[\Sigma^\iota; \Sigma^o; \overline{C}; C]}\ \Delta \vdash C \leq C'$$

$$\text{(K-WRAP-T)}\ \frac{\begin{array}{l}\Delta; \emptyset; x_1{:}C_1 \ldots x_n{:}C_n \vdash e_i{:}C_i''\ \forall i \in 1..k \\ \Delta \vdash CE{:}[\Sigma^\iota; \Sigma^o; C_1' \ldots C_k'; C]\end{array}}{\begin{array}{l}\Delta \vdash\ CE[\mathtt{K}(C_1\ x_1 \ldots C_n\ x_n)\{e_1 \ldots e_k\}]{:} \\ \qquad [\Sigma^\iota; \Sigma^o; C_1 \ldots C_n; C]\end{array}}\ \begin{array}{l}\Delta \vdash C_i'' \leq C_i' \\ \forall i \in 1..k\end{array}$$

Fig. 5. Typing rules for programs and class expressions

$$\text{(VAR-T)}\ \frac{}{\Delta;\Gamma;\Pi \vdash x{:}C}\ \Pi(x) = C \qquad \text{(CLIENT-FIELD-T)}\ \frac{\Delta;\Gamma;\Pi \vdash e_0{:}C_0}{\Delta;\Gamma;\Pi \vdash e_0.F{:}C}\ mtype(\Delta, C_0, F) = C$$

$$\text{(CLIENT-INVK-T)}\ \frac{\Delta;\Gamma;\Pi \vdash e_0{:}C_0 \quad \Delta;\Gamma;\Pi \vdash e_i{:}C_i'\ \forall i \in 1..n}{\Delta;\Gamma;\Pi \vdash e_0.M(e_1 \dots e_n){:}C}\ \begin{array}{l} mtype(\Delta, C_0, M) = C_1 \dots C_n \to C \\ \Delta \vdash C_i' \le C_i\ \forall i \in 1..n \end{array}$$

$$\text{(INT-FIELD-T)}\ \frac{}{\Delta;\Gamma;\Pi \vdash f{:}C}\ \Gamma(f) = C$$

$$\text{(INT-INVK-T)}\ \frac{\Delta;\Gamma;\Pi \vdash e_i{:}C_i'\ \forall i \in 1..n}{\Delta;\Gamma;\Pi \vdash m(e_1 \dots e_n){:}C}\ \begin{array}{l} \Gamma(m) = C_1 \dots C_n \to C \\ \Delta \vdash C_i' \le C_i\ \forall i \in 1..n \end{array}$$

$$\text{(NEW-T)}\ \frac{\Delta;\Gamma;\Pi \vdash e_i{:}C_i'\ \forall i \in 1..n}{\Delta;\Gamma;\Pi \vdash \texttt{new}\ C(e_1 \dots e_n){:}C}\ \begin{array}{l} \Delta(C) = [\emptyset; \_; C_1 \dots C_n; \_] \\ \Delta \vdash C_i' \le C_i\ \forall i \in 1..n \end{array}$$

Fig. 6. Typing rules for expressions

(pre)object's class in Δ^r. Rules for other forms of expressions are analogous to those in Fig. 6, plus propagation of the runtime class type environment.

$$\text{(BLOCK-T)}\ \frac{\Delta;\Delta^r;\Gamma;\Pi \vdash v{:}C' \quad \Delta;\Delta^r;\Gamma';C' \vdash \overline{\mu}{:}\Gamma^{\overline{\mu}} \quad \Delta;\Delta^r;\Gamma';\Pi \vdash e{:}C}{\Delta;\Delta^r;\Gamma;\Pi \vdash [\overline{\mu}; v \mid e]{:}C}\ \Gamma' = \Gamma, \Delta^r(C'), \Gamma^{\overline{\mu}}$$

$$\text{(PRE-OBJ-T)}\ \frac{\Delta;\Delta^r;\Gamma;\Pi \vdash e_i{:}C_i'\ \forall i \in 1..n}{\Delta;\Delta^r;\Gamma;\Pi \vdash C(f_1 = e_1; \dots f_n = e_n;){:}C}\ \begin{array}{l} \Delta^r(C) = f_1{:}C_1 \dots f_n{:}C_n \\ \Delta \vdash C_i' \le C_i\ \forall i \in 1..n \end{array}$$

Fig. 7. Typing rules for runtime expressions

Soundness of the type system is expressed by the following theorems.

Theorem 1 (Soundness w.r.t. flattening relation). *If* $\vdash p{:}\Delta$, *then* $p \xrightarrow{\star} p'$ *for some* p' *flat program, and* $\vdash p'{:}\Delta$.

Let us denote by Δ_p^r the runtime class type environment extracted from a flat program p. That is, for each basic class declaration of form $C \mapsto [\emptyset \mid o \mid \{\tau\ \overline{\varphi}\ \kappa\ \overline{\mu}\}]$ in p, $\Delta_p^r(C) = \Gamma^{\overline{\varphi}}$.

Theorem 2 (Progress). *If* $\vdash p{:}\Delta$, *then* $\Delta;\Delta_p^r;\emptyset;\emptyset \vdash e{:}C$ *implies that either* e *is a value or* $e \longrightarrow_p e'$ *for some* e'.

Theorem 3 (Subject reduction). *If* $\vdash p{:}\Delta$, *then* $\Delta; \Delta_p^r; \Gamma; \Pi \vdash e{:}C$ *and* $e \longrightarrow_p e'$ *imply that* $\Delta; \Delta_p^r; \Gamma; \Pi \vdash e'{:}C'$, *and* $\Delta \vdash C' \leq C$.

Proof of Th. 1 is is a simple adaptation from [3], others can be found in [16].

4 Direct semantics

Direct semantics allows a modular approach where each class (module) can be analyzed (notably, compiled) in isolation, since references to other classes do not need to be resolved before runtime. In this case, look-up is a non trivial procedure where a class member (e.g., method) is retrieved from other classes and possibly modified as effect of the module operators. Since FJIG subsumes a variety of mechanisms for class composition, including standard inheritance, mixins, traits, and hiding, the definition of direct semantics for FJIG provides a guideline which can be emulated by real extensions of class-based languages, and also a hint to implementation.

In order to give this definition, block expressions are generalized as shown in the top section of Fig. 8. That is, besides the previous components, a block contains a *path map* $\hat{\iota}$ which maps internal names to *paths* π, which denote a subterm in the class expression defining the class C of the current object (an implementation could use a pointer). More precisely, a path π always denotes a subterm of the form $\mathit{freeze}_N CE$, and is used as a permanent reference to the definition of member N in CE. Indeed, the external name N can be changed or removed by effect of outer reduct operators; however, references via π are not affected. Hence, when a reference π is encountered during current method execution, lookup of N in CE is triggered (see more explanations below). In flattening semantics, C is always a basic class, hence this case never happens.

The center section of the figure contains the new rules for expression reduction. When a member reference (external name or path) $\hat{N}$ needs to be resolved, the lookup procedure starts the search of $\hat{N}$ from receiver's class C and, if successful, returns a corresponding internal name inside a block expression, as shown in rules (CLIENT-FIELD) and (CLIENT-INVK). In flattening semantics, C is always a basic class, hence lookup is trivial and the side condition can be equivalently expressed as in the analogous rules in Fig. 3.

When an internal name n is encountered, it is either directly mapped to a definition, or to a path. The former case happens when n was a local name in the basic class containing the definition of the method which is currently being executed. In this case, the corresponding definition is taken, as shown in rules (INT-FIELD) and (INT-INVK). The latter case happens when n was an abstract or virtual name inside the basic class containing the definition of the method which is currently executed, and n has been permanently bound to some definition by an outer freeze operator (recall that only classes where all members are frozen can be instantiated). In this case, lookup of this definition is started from receiver's class via the path π, and, if successful, the internal name n is replaced by the name n' found by lookup; moreover, the corresponding path map and

$\pi ::= i_1 \dots i_k$ path ($i \in \{1,2\}$)

$\hat{N} ::= N \mid \pi$ member reference (external name or path)

$\hat{\iota} ::= n_1 \mapsto \pi_1 \dots n_k \mapsto \pi_k$ path map

$e ::= \dots \mid [\hat{\iota}; \overline{\mu}; v \mid e]$ (generalized) block

(CLIENT-FIELD) $$\frac{}{v^C.F \longrightarrow_p [\hat{\iota}; \overline{\mu}; v^C \mid f]} \; lookup_p\langle F, C\rangle = [\hat{\iota}; \overline{\mu} \mid f]$$

(CLIENT-INVK) $$\frac{}{v^C.M(\overline{v}) \longrightarrow_p [\hat{\iota}; \overline{\mu}; v^C \mid m(\overline{v})]} \; lookup_p\langle M, C\rangle = [\hat{\iota}; \overline{\mu} \mid m]$$

(INT-FIELD) $$\frac{}{[\hat{\iota}; \overline{\mu}; v \mid \mathcal{E}\{f\}] \longrightarrow_p [\hat{\iota}; \overline{\mu}; v \mid \mathcal{E}\{v_i\}]} \; \begin{array}{l} f \notin HB(\mathcal{E}) \\ v = C(f_1{=}v_1 \dots f_n{=}v_n) \\ f{=}f_i \end{array}$$

(INT-INVK) $$\frac{}{[\hat{\iota}; \overline{\mu}; v \mid \mathcal{E}\{m(\overline{v})\}] \longrightarrow_p [\hat{\iota}; \overline{\mu}; v \mid \mathcal{E}\{e[\overline{v}/\overline{x}][v^C/\texttt{this}]\}]} \; \begin{array}{l} m \notin HB(\mathcal{E}) \\ \overline{\mu}(m) = \langle \overline{x}, e\rangle \end{array}$$

(PATH) $$\frac{}{[\hat{\iota}, n \mapsto \pi; \overline{\mu}; v^C \mid e] \longrightarrow_p [\hat{\iota}, \hat{\iota}'; \overline{\mu}[n'/n], \overline{\mu}'; v^C \mid e[n'/n]]} \; \begin{array}{l} n \in names(e) \\ lookup_p\langle \pi, C\rangle = \\ [\hat{\iota}'; \overline{\mu}' \mid n'] \end{array}$$

(OBJ-CREATION) $$\frac{}{\texttt{new}\ C(\overline{v}) \longrightarrow_p C(\overline{f{=}e[\overline{v}/\overline{x}]})} \; \begin{array}{l} k\text{-}lookup_p(C) = \texttt{K}(C_1\ x_1 \dots C_n\ x_n)\{\overline{f{=}e}\} \\ \overline{x}{=}x_1 \dots x_n \end{array}$$

(EXIT-BLOCK) $$\frac{}{[\hat{\iota}; \overline{\mu}; v \mid e] \longrightarrow_p e} \; names(e){=}\emptyset$$

$lookup_p\langle \hat{N}, \pi, C\rangle = lookup_p\langle \hat{N}, \pi, CE\rangle$
  if $p(C) = CE$
$lookup_p\langle N, \pi, [\iota \mid o, N{\mapsto}n \mid \{\tau\ \overline{\varphi}\ \kappa\ \overline{\mu}\}]\rangle = [\iota; \emptyset; \overline{\mu} \mid n]$
$lookup_p\langle \hat{N}, \pi, CE_1 + CE_2\rangle = \alpha_i([\iota; \hat{\iota}; \overline{\mu} \mid n])$
  if $lookup_p\langle \hat{N}, \pi.i, CE_i\rangle = [\iota; \hat{\iota}; \overline{\mu} \mid n], i \in \{1,2\}$
$lookup_p\langle \hat{N}, \pi, {}_{\sigma^\iota|}CE_{|\sigma^o}\rangle = [\sigma^\iota \circ \iota; \hat{\iota}; \overline{\mu} \mid n]$
  if $lookup_p\langle \hat{N}', \pi.1, CE\rangle = [\iota; \hat{\iota}; \overline{\mu} \mid n]$,
  $\hat{N}' = \sigma^o(N)$ if $\hat{N} = N$, $\hat{N}' = \hat{N}$ otherwise
$lookup_p\langle \hat{N}, \pi, freeze_N CE\rangle = [\iota; \hat{\iota}, n_1{\mapsto}\pi \dots n_k{\mapsto}\pi; \overline{\mu} \mid n]$
  if $\hat{N} \neq \pi$, $N \notin img(\iota)$,
  $lookup_p\langle \hat{N}, \pi.1, CE\rangle = [\iota, n_1{\mapsto}N \dots n_k{\mapsto}N; \hat{\iota}; \overline{\mu} \mid n]$
$lookup_p\langle \pi, \pi, freeze_N CE\rangle = [\iota; \hat{\iota}, n_1{\mapsto}\pi \dots n_k{\mapsto}\pi; \overline{\mu} \mid n]$
  if $N \notin img(\iota)$,
  $lookup_p\langle N, \pi.1, CE\rangle = [\iota, n_1{\mapsto}N \dots n_k{\mapsto}N; \hat{\iota}; \overline{\mu} \mid n]$
$lookup_p\langle \hat{N}, \pi, CE[\texttt{TT}{\leq}C]\rangle = lookup_p\langle \hat{N}, \pi.1, CE\rangle$
$lookup_p\langle \hat{N}, \pi, CE[\texttt{K}(\overline{C\ x})\{\overline{e}\}]\rangle = lookup_p\langle \hat{N}, \pi.1, CE\rangle$

$k\text{-}lookup_p(C) = k\text{-}lookup_p(CE)$
  if $p(C) = CE$
$k\text{-}lookup_p([\emptyset \mid o \mid \{\tau\ \overline{\varphi}\ \kappa\ \overline{\mu}\}]) = \kappa$
$k\text{-}lookup_p(CE_1 + CE_2) = \texttt{K}(\overline{C\ x})\{\alpha_1(\overline{f = e}), \alpha_2(\overline{f' = e'})\}$
  if $k\text{-}lookup_p(CE_1) = \texttt{K}(\overline{C\ x})\{\overline{f = e}\}$,
  $k\text{-}lookup_p(CE_2) = \texttt{K}(\overline{C\ x})\{\overline{f' = e'}\}$
$k\text{-}lookup_p({}_{\sigma^\iota|}CE_{|\sigma^o}) = k\text{-}lookup_p(CE)$
$k\text{-}lookup_p(freeze_N CE) = k\text{-}lookup_p(CE)$
$k\text{-}lookup_p(CE[\texttt{TT}{\leq}C]) = k\text{-}lookup_p(CE)$
$k\text{-}lookup_p(CE[\texttt{K}(\overline{\texttt{D y}})\{\overline{e}\}]) = \texttt{K}(\overline{\texttt{D y}})\{\overline{f{=}e[\overline{e}/\overline{x}]}\}$
  if $\overline{x} = x_1 \dots x_n$,
  $k\text{-}lookup_p(CE) = \texttt{K}(C_1\ x_1 \dots C_n\ x_n)\{\overline{f{=}e}\}$

Fig. 8. Direct semantics

methods are merged with the original ones (α-renaming can be used to avoid conflicts among internal names in this phase). This is shown in rule (PATH). In flattening semantics, the latter case never happens, hence only the first two rules are needed.

Creation of an instance of class, say, C, also involves a *constructor lookup* procedure, which returns, starting from class C, the appropriate constructor, by retrieving and possibly modifying constructors of other classes (this generalizes what happens in standard Java-like languages, where the superclass constructor is always invoked). In flattening semantics, C is always a basic class, hence constructor lookup is trivial and the side condition can be equivalently expressed as in the corresponding rule in Fig. 3.

Lookup and constructor lookup are defined in the bottom section of the figure.

The lookup procedure is modeled by a function which, given a program p, takes three more arguments: a member reference (external name or path) $\hat{N}$, a path π, which acts as an accumulator and keeps track of the current subterm of the class expression which is examined, and a class name C. When lookup is started, π is always the empty path Λ, and $\mathit{lookup}_p\langle \hat{N}, \Lambda, C\rangle$ is abbreviated by $\mathit{lookup}_p\langle \hat{N}, C\rangle$.

The lookup function returns a triple consisting of input map, path map, methods and an internal name, written $[\iota; \hat{\iota}; \overline{\mu} \mid n]$. However, the final result of lookup (that is, the result returned for the initial call) is expected to be always of form $[\emptyset; \hat{\iota}; \overline{\mu} \mid n]$, abbreviated by $[\hat{\iota}; \overline{\mu} \mid n]$, since all abstract/virtual internal names are expected to be eventually bound to a path as effect of some freeze operator.

The first two clauses defining lookup are trivial and state that looking for a member reference starting from a class name C means looking in the definition of C, and that looking for an external name N in a basic class only succeeds if the name is present in the class, and returns the corresponding input map, methods and internal name. Note that the case where we look for a path π in a basic class is expected to never happen.

The third clause defines lookup on a sum expression. In this case, lookup is propagated to both arguments. This definition is a priori non-deterministic, but is expected to be deterministic on class expressions which can be safely flattened, since in this case an external name cannot be found on both sides. For member references which are paths, instead, determinism is guaranteed by construction since the path exactly corresponds to a subterm. In case lookup succeeds on one of the two arguments, the result is modified by renaming field local names in a way which keeps track of this argument. For instance, if lookup succeeded on the first argument, then every field internal name f is renamed to $f.1$. This renaming is denoted by α_i. We choose this canonical α-renaming for concreteness, but any other could be chosen, provided that it is consistent with that in constructor lookup.

For instance, let us consider the following program[10]:

[10] In order to write more readable examples, we assume integer values and operations, and omit default constructor and `ThisType` constraint.

$C \mapsto\ C_1 + C_2$
$C_1 \mapsto [\emptyset \,|\, \ldots \,|\, \{\ \texttt{int}\ f; \texttt{K}()\{f = 3\}\ \ldots\}]$
$C_2 \mapsto [\emptyset \,|\, \ldots, M \mapsto m \,|\, \{\ \texttt{int}\ f; \texttt{K}()\{f = 5\}\ \texttt{int}\ m()\{\texttt{return}\ f + 1;\}\}]$

and the expression $\texttt{new}\ C().M()$. An instance of class C has two fields, inherited from C_1 and C_2, and initialized to 3 and 5, respectively. They are both named f in the original classes; however, they are renamed during constructor lookup (see the clause for sum), hence the above expression reduces to $C(f.1{\mapsto}3, f.2{\mapsto}5).M()$. Now, M is invoked, starting the lookup from C, and the search is propagated to both C_1 and C_2. Only the lookup in C_2 is successful and returns the result

$$[;;\texttt{int}\ m()\{\texttt{return}\ f + 1;\} \,|\, m]$$

which is modified in $[;;\texttt{int}\ m()\{\texttt{return}\ f.2{+}1;\} \,|\, m]$ to take into account that the method has been found in the second argument. Hence, this method invocation reduces to $[;\texttt{int}\ m()\{\texttt{return}\ f.2 + 1;\}; C(f.1{\mapsto}3, f.2{\mapsto}5) \,|\, m]$ where the body of m correctly refers to the second field.
In flattening semantics, C reduces to the following basic class:

$[\emptyset \,|\, \ldots, M \mapsto m \,|\, \{\ \texttt{int}\ f.1; \texttt{int}\ f.2; \kappa\ \texttt{int}\ m()\{\texttt{return}\ f.2 + 1;\} \ldots\}]$
$\kappa = \texttt{K}()\{f.1 = 3, f.2 = 5\}$

Note that here the clash between the two fields is resolved during flattening (hence before runtime), by α-renaming. We have chosen as α-renaming the same used in direct semantics as an help for the reader, but of course in this case any other arbitrary α-renaming would work as well.
The fourth clause defines lookup on a reduct expression. In this case, lookup of an external name is propagated under the name the member has in the argument, given by the output renaming σ^o. Instead, lookup of a path is simply propagated, since paths are permanent references which are not affected by renamings. Moreover, the result of lookup on the argument must be modified to ensure that internal names refer to the appropriate external names obtained via the input renaming σ^ι.
For instance, consider a program including

$C \mapsto {}_{M_1 \mapsto M_1'|} C'_{|M \mapsto M'}$
$C' \mapsto [m' \mapsto M_1 \,|\, M' \mapsto m \,|\, \{\ \ldots\ \texttt{int}\ m()\{\texttt{return}\ m'();\}\}]$

and assume that some method invocation triggers the lookup for M in C. Then, the lookup is propagated under the name M' to C'. The lookup of M' in C' is successful and returns the result $[m' \mapsto M_1;;\texttt{int}\ m()\{\texttt{return}\ m'();\} \,|\, m]$ which is modified in $[m' \mapsto M_1';;\texttt{int}\ m()\{\texttt{return}\ m'();\} \,|\, m]$ as an effect of the input renaming.
In flattening semantics, C reduces to the following basic class:

$$[m' \mapsto M_1' \,|\, M \mapsto m \,|\, \{\ \ldots\ \texttt{int}\ m()\{\texttt{return}\ m'();\}\}]$$

There are two clauses defining lookup on a freeze expression. The former handles most cases, except the special situation in which we are exactly looking for the member that has been frozen in the current subterm π, which has the form

$freeze_N CE$. In this special case (second clause) the lookup of N in CE is triggered. Moreover, the result is modified, since internal names referring to N must now refer to the permanent reference π. Otherwise (first clause), the lookup is propagated, and the result of the lookup on the argument is modified as in the previous case.
Consider the program

$$C \mapsto \ freeze_F C'$$
$$C' \mapsto [f \mapsto F \,|\, F \mapsto f', M \mapsto m \,|\, \{\ \texttt{int}\ f'; \texttt{K}()\{f' = 42\}\ \texttt{int}\ m()\{\texttt{return}\ f + 1;\}\}]$$

and the expression $\texttt{new}\ C().M()$.
An instance of class C has one field, inherited from C' and initialized to 42. Hence, the above expression reduces to $C(f' \mapsto 42).M()$. Now, M is invoked, starting the lookup from C, and the search is propagated to C'. The lookup in C' is successful and returns the result $[f \mapsto F; ; \texttt{int}\ m()\{\texttt{return}\ f + 1;\} \mid m]$, which is modified in $[; f \mapsto \Lambda; \texttt{int}\ m()\{\texttt{return}\ f + 1;\} \mid m]$, where Λ denotes the empty path, to take into account that F has been frozen. Hence, the method invocation reduces to $[f \mapsto \Lambda; \texttt{int}\ m()\{\texttt{return}\ f + 1;\}; C(f \mapsto 42) \mid m]$, where the body of m correctly refers to F frozen in the top level freeze.
In flattening semantics, C reduces to the following basic class:

$$[\emptyset \,|\, F \mapsto f', M \mapsto m \,|\, \{\ \texttt{int}\ f'; \texttt{K}()\{f' = 42\}\ \texttt{int}\ m()\{\texttt{return}\ f' + 1;\}\}]$$

Fig. 9 shows a more involved example comparing flattening and direct semantics. The top section of the figure lists some abbreviations, the second shows the four classes composing program p. Class A defines a method M whose body invokes the abstract method M'. Class B has a local field f initialized to 0 and defines a method M' which returns this field. Class C is obtained by summing A and B, and then freezing method M'. Finally, class D is obtained by hiding method M' in C (in the reduct, the input renaming is empty since there are no input names, and the output renaming maps "no new name" into M' and is the identity on M) and then summing a new definition for M'. The following three sections of the figure shows how the class expressions for C and D are reduced, the resulting flat program p' and the reduction of expression $\texttt{new}\ D().M()$ in the context of p'. Finally, the last section shows direct semantics of the same expression in the context of p.
The example shows how the method originally called M' in B is correctly invoked via the path 1.1, even though M' has been hidden and then replaced by an homonymous method.
The following theorem states that flattening is equivalent to direct semantics. We denote by $\xrightarrow{\star}$ the reflexive and transitive closure of the flattening relation, and analogously for the reduction relation. The proof can be found in [17].

Theorem 4. *If $p \xrightarrow{\star} p'$, and e is an expression with no paths, then $e \xrightarrow{\star}_p v$ iff $e \xrightarrow{\star}_{p'} v$.*

$\mu_A \equiv C\ m()\{\texttt{return}\ m'();\}$
$\overline{\mu}_{sum} \equiv \mu_A\ C\ m''()\{\texttt{return}\ f;\}$
$\overline{\mu}_C \equiv C\ m()\{\texttt{return}\ m''();\}\ C\ m''()\{\texttt{return}\ f;\}$
$\overline{\mu}_D \equiv \overline{\mu}_C\ C\ m'()\{\texttt{return}\ 8;\}$
$\mu'' \equiv C\ m''()\{\texttt{return}\ f.2.1;\}$

$$\begin{aligned} p \equiv\ & A = [m' \mapsto M' \,|\, M \mapsto m \,|\, \{\mu_A\}] \\ & B = [\emptyset \,|\, M' \mapsto m' \,|\, \{\ C\ f;\ \texttt{K}()\{f = 0\}\ C\ m'()\{\texttt{return}\ f;\}\}] \\ & C = \mathit{freeze}_{M'}(A + B) \\ & D = {}_{\emptyset|}C_{|\_ \mapsto M', M \mapsto M} + [\emptyset \,|\, M' \mapsto m' \,|\, \{C\ m'()\{\texttt{return}\ 8;\}]\} \end{aligned}$$

$\mathit{freeze}_{M'}(A + B) \longrightarrow$
$\mathit{freeze}_{M'}[m' \mapsto M' \,|\, M \mapsto m, M' \mapsto m'' \,|\, \{\ C\ f;\ \texttt{K}()\{f = 0\}\ \overline{\mu}_{sum}\}] \longrightarrow$
$[\emptyset \,|\, M \mapsto m, M' \mapsto m'' \,|\, \{\ C\ f;\ \texttt{K}()\{f = 0\}\ \overline{\mu}_C\}]$

$D \xrightarrow{\star}$
$[\emptyset \,|\, M \mapsto m \,|\, \{\ C\ f;\ \texttt{K}()\{f = 0\}\ \overline{\mu}_C\}] + [\emptyset \,|\, M' \mapsto m' \,|\, \{C\ m'()\{\texttt{return}\ 8;\}]\} \longrightarrow$
$[\emptyset \,|\, M \mapsto m, M' \mapsto m' \,|\, \{\ C\ f;\ \texttt{K}()\{f = 0\}\ \overline{\mu}_D\}]$

$$\begin{aligned} p' \equiv\ & A = [m' \mapsto M' \,|\, M \mapsto m \,|\, \{\mu_A\}] \\ & B = [\emptyset \,|\, M' \mapsto m' \,|\, \{\ C\ f;\ \texttt{K}()\{f = 0\}\ C\ m'()\{\texttt{return}\ f;\}\}] \\ & C = [\emptyset \,|\, M \mapsto m, M' \mapsto m'' \,|\, \{\ C\ f;\ \texttt{K}()\{f = 0\}\ \overline{\mu}_C\}] \\ & D = [\emptyset \,|\, M \mapsto m, M' \mapsto m' \,|\, \{\ C\ f;\ \texttt{K}()\{f = 0\}\ \overline{\mu}_D\}] \end{aligned}$$

$\texttt{new}\ D().M() \longrightarrow_{p'} D(f = 0).M() \longrightarrow_{p'} [\overline{\mu}_D; D(f = 0) \,|\, m()] \longrightarrow_{p'}$
$[\overline{\mu}_D; D(f = 0) \,|\, m''()] \longrightarrow_{p'} [\overline{\mu}_D; D(f = 0) \,|\, f] \longrightarrow_{p'} [\overline{\mu}_D; D(f = 0) \,|\, 0] \longrightarrow_{p'} 0$

$\texttt{new}\ D().M() \longrightarrow_p$
$D(f.2.1 = 0).M() \longrightarrow_p$
$[m' \mapsto 1.1; \mu_A; D(f.2.1 = 0) \,|\, m()] \longrightarrow_p$
$[m' \mapsto 1.1; \mu_A; D(f.2.1 = 0) \,|\, m'()] \longrightarrow_p$
$[m' \mapsto 1.1; \mu_A, \mu''; D(f.2.1 = 0) \,|\, m''()] \longrightarrow_p$
$[m' \mapsto 1.1; \mu_A, \mu''; D(f.2.1 = 0) \,|\, f.2.1] \longrightarrow_p$
$[m' \mapsto 1.1; \mu_A, \mu''; D(f.2.1 = 0) \,|\, 0] \longrightarrow_p$
0

$\mathit{k\text{-}lookup}_p(D) = \texttt{K}()\{f.2.1 = 0\}$
$\mathit{lookup}_p\langle M, \Lambda, D\rangle = [\Lambda; m' \mapsto 1.1; \mu_A | m]$

$\mathit{lookup}_p\langle 1.1, \Lambda, D\rangle = [\Lambda; \Lambda; \mu'' | m'']$

Fig. 9. Example

5 Conclusion

We have presented FJig, a core calculus which formalizes the Bracha's Jigsaw framework [8] in a Java-like setting. The design of FJig comes out naturally, yet not trivially, by taking Featherweight Java [15] as starting point and replacing inheritance by the more general composition operators of Jigsaw.

We believe that such a core calculus can be useful for many research directions. First, it provides a simple unifying formalism for encoding and comparing a large variety of different mechanisms for software composition in class-based languages, including standard inheritance, mixin classes, traits and hiding. Then, it can serve as the basis for the design of a real language based on Jigsaw principles. Moreover, it could be enriched by behavioural types, leading to a class-based specification language, in the spirit of, e.g., JML [18], allowing modular development and composition of class specifications.

We have also defined two different execution models for the calculus, flattening and direct semantics, and proved their equivalence. That is, we have shown the equivalence of two different views on inheritance in a formal setting with a more sophisticated composition mechanism, where, e.g., mixin classes and traits can be subsumed. This can also greatly help in integrating such features, or other modularity mechanisms, in standard class-based languages, since it gives practical hints on implementation.

Apart from the two key references mentioned above, this work has been directly influenced by work on traits [20,10], mostly by the recent developments [19,6,7]. In particular, we share with [6,7] the objective of replacing inheritance by more flexible operators. Concerning flattening and direct semantics, the most direct source of inspiration for our work has been [19], which defines a direct semantics for traits. Essentially, their dynamic look-up algorithm can be seen as a simplified version, handling sum and output reduct only, of ours.

The focus of this paper is on providing a simple and compact model for a language based on the Jigsaw framework in a Java-like setting, hence we have only outlined in Sect. 1 a simple surface language. As mentioned above, we leave to further work a deeper investigation of a realistic language design, and a more precise analysis on how different mechanisms such as standard inheritance, mixin classes, traits can be encoded into FJig. We also plan to develop a prototype; a very preliminary interpreter of flattening semantics, assigned as master thesis, can be found at `http://www.disi.unige.it/person/LagorioG/FJig/`. We also plan to investigate smart implementation techniques of direct semantics in the prototype interpreter.

References

1. D. Ancona, G. Lagorio, and E. Zucca. Jam–designing a Java extension with mixins. *ACM Transactions on Programming Languages and Systems*, 25(5):641–712, September 2003.

2. D. Ancona and E. Zucca. Overriding operators in a mixin-based framework. In *PLILP '97 - 9th Intl. Symp. on Programming Languages, Implementations, Logics and Programs*, LNCS 1292. Springer, 1997.
3. D. Ancona and E. Zucca. A calculus of module systems. *Journ. of Functional Programming*, 12(2):91–132, 2002.
4. A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits. In *Advances in Smalltalk - 14th International Smalltalk Conference (ISC 2006)*, volume 4406. Springer, 2007.
5. A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. *Comput. Lang. Syst. Struct.*, 34(2-3):83–108, 2008.
6. V. Bono, F. Damiani, and E. Giachino. Separating type, behavior, and state to achieve very fine-grained reuse. In *9th Intl. Workshop on Formal Techniques for Java-like Programs*, 2007.
7. V. Bono, F. Damiani, and E. Giachino. On traits and types in a Java-like setting. In *TCS'08 - IFIP Int. Conf. on Theoretical Computer Science*. Springer, 2008.
8. G. Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.
9. D. Duggan and C. Sourelis. Mixin modules. In *Intl. Conf. on Functional Programming 1996*. ACM Press, 1996.
10. K. Fisher and J. Reppy. A typed calculus of traits. In *FOOL'04 - Intl. Workshop on Foundations of Object Oriented Languages*, 2004.
11. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *ACM Symp. on Principles of Programming Languages 1998*. ACM Press, 1998.
12. E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.
13. T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In *ESOP 2002 - European Symposium on Programming 2002*, LNCS 2305. Springer, 2002.
14. T. Hirschowitz, X. Leroy, and J. B. Wells. Call-by-value mixin modules: Reduction semantics, side effects, types. In *ESOP 2003 - European Symposium on Programming 2003*, LNCS 2986. Springer, 2004.
15. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
16. G. Lagorio, M. Servetto, and E. Zucca. Featherweight Jigsaw - a minimal core calculus for modular composition of classes. Technical report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, December 2008. Full version.
17. G. Lagorio, M. Servetto, and E. Zucca. Flattening versus direct semantics for Featherweight Jigsaw. In *FOOL'09 - Intl. Workshop on Foundations of Object Oriented Languages*, 2009.
18. G. T. Leavens. Tutorial on JML, the Java modeling language. In *Automated Software Engineering (ASE 2007)*. ACM Press, 2007.
19. L. Liquori and A. Spiwack. FeatherTrait: A modest extension of Featherweight Java. *ACM Transactions on Programming Languages and Systems*, 30(2), 2008.
20. N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *ECOOP'03 - Object-Oriented Programming*, LNCS 2743. Springer, 2003.
21. J. B. Wells and R. Vestergaard. Confluent equational reasoning for linking with first-class primitive modules. In *ESOP 2000 - European Symposium on Programming 2000*, LNCS 1782. Springer, 2000.