

Flattening versus direct semantics for Featherweight Jigsaw^{*}

Giovanni Lagorio Marco Servetto Elena Zucca
DISI, Univ. of Genova, v. Dodecaneso 35, 16146 Genova, Italy
email: {lagorio, servetto, zucca}@disi.unige.it

Abstract

Inheritance in object-oriented languages allows, roughly, to obtain the same effect one would get by duplicating the methods of the parent class in the heir. However, the key advantage is that source code duplication is avoided, and the code of the parent is, instead, found on demand, through a runtime procedure called *method look-up*. In other words, two different semantics of inheritance can be given: *flattening* semantics, that is, by translation into a language with no inheritance, and *direct* semantics, that is, by formalizing dynamic method look-up. Analogously, many other composition mechanisms which have been proposed for enhancing the object-oriented paradigm, such as mixins and traits, can be formally defined either by translation into standard inheritance, or by a providing a direct execution model.

Flattening semantics generally provides a simpler model and can be used as a guide in language design. However, it is not adequate for compositional analysis since the binary code for each code fragment, say, a class, can be generated only when all used fragments are available.

In this paper we define both semantics and prove their equivalence for Featherweight Jigsaw, a class-based language providing a very general framework for software composition, subsuming, besides other mechanisms, standard inheritance, mixins, and traits.

Introduction

Inheritance in object-oriented languages allows, roughly, to obtain the same effect one would get by duplicating the methods of the parent class in the heir. However, the key advantage is that source code duplication is avoided, and the code of the parent is, instead, found on demand, through a runtime procedure called *method look-up*. In other words, two different semantics of inheritance can be given: either by translation into a language with no inheritance, or by providing a direct execution model which formalizes dynamic method look-up. This difference is analogous, in a sense, to the one between using compilation or interpretation for implementing

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

a language, and has been often mentioned in the literature under different names. For instance, in [4] the former is called “copy semantics”, and the fact that the two semantics should be equivalent is used as a guideline to discuss and evaluate different rules for overloading resolution. This terminology has been followed in [6, 7, 8] for explaining semantics of multimethods. In [2] the design of an extension of Java with mixin classes is guided by the “copy principle”, that is, that the behaviour of a class obtained by mixin instantiation should be the same that one would get by copying the code of the mixin into the heir. In [13] semantics of inner classes in Java is given both by translation and by direct reduction rules, and a proof of equivalence is provided.¹

In particular, many composition mechanisms which have been proposed for enhancing the object-oriented paradigm can be formally defined either by translation into standard inheritance, or by a providing a direct execution model. For instance, semantics of traits [18, 10] has been always given in the literature, with the notable exception of [16, 15], by translation into a language without traits, hence by flattening style, whereas for mixin classes there are examples of both solutions [2, 11].

In this paper, we call the two kinds of semantics *flattening* and *direct semantics*, respectively. Both are, in our opinion, useful. Flattening semantics usually provides a simple and intuitive model of what is expected to happen, and can be even used as a guide in language design, as it is the case in [4, 2] mentioned above. On the other hand, it can be inadequate for compositional analysis, since the binary code for each fragment, say, a class, can be generated only when all used fragments are available. Direct semantics, instead, allows to analyze a class in isolation, since the references to members provided by other classes do not need to be resolved at compile time. These references are indeed resolved at runtime through a dynamic look-up procedure. This procedure is a natural generalization of dynamic look-up for the standard inheritance case, where the method is recursively searched in the parent(s) class(es) if not present. However, this generalization can be non-trivial, leading to a more difficult understanding of the language execution model. Clearly, the ideal situation would be to have both semantics and know that they are equivalent.

In this paper, we achieve this result in a very general framework, that is, Featherweight Jigsaw (shortly FJIG), a class-based calculus formalizing, in a Java-like setting, the Jigsaw framework originally proposed in Bracha’s seminal work [9]. The full presentation of FJIG, including the type system and the related soundness proof, can be found in [14]. Note that solving the problem for FJIG is

¹The well-known difference between *heterogeneous* and *homogeneous* translation of generics (see, e.g., [17]) is only partly analogous, since in this case both solutions are translations, that is, the runtime model does not change.

much more than a case-study, since this calculus subsumes a variety of mechanisms for composition of classes, including standard inheritance, mixins, traits, and hiding. Hence, the two equivalent semantics presented here can serve as a paradigmatic solution.

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

1. An informal introduction

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

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.

In this paper, we do not illustrate typing issues of FJIG, since they are not relevant for our aim. Hence, we also slightly simplify the syntax omitting a few constructs that only play the role of type annotations. We refer to [14] for a full presentation.

A program consists of a sequence of *class declarations* (class name and class expression), as in FJ. 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 are basic classes, class names, 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 as 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 it is usually required for standard inheritance.

A basic class consists of a sequence of field declarations, a constructor declaration, and a sequence of method declarations. We assume that no field or method is declared twice and order is immaterial.

Field and method declarations are as in FJ, except that they are decorated by one of the following *modifiers*: `abstract`, `virtual`, `frozen` or `local`, and 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 as the effect of the composition operators. However, redefinition of a `frozen` member does not affect other members, which still refer to its original definition.
- Finally, as the name suggests, a `local` member cannot be selected by a client, and is not affected by composition operators, hence its definition cannot be changed.

We assume by default (hence omit) the modifier `frozen` for fields and `virtual` for methods. A class having at least one `abstract` member must be declared `abstract`.

The following example illustrates the meaning of modifiers.²

```
abstract class A {
  abstract int M1();
}
```

²To write more readable examples, we assume that the primitive type `int` and its operations are available.

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

These two classes are abstract (hence cannot be instantiated), have no fields, and the default constructor (see below). A concrete class can be obtained by using 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 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 following 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:

```
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.

The above modifiers are rather intuitive and well-established for methods, whereas, to the best of our knowledge, they have been never applied to fields. The meaning is analogous, 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; }
}
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 one³ 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,

³Since overloading is not allowed.

p	$::= \overline{cd}$	program	B	$::= \{\overline{\varphi} \kappa \overline{\mu}\}$	basic class
cd	$::= cmod \text{ class } C \ CE$	class declaration	φ	$::= mod \ C \ F;$	field
$cmod$	$::= \text{abstract} \mid \epsilon$	class modifier	κ	$::= kh\{\overline{F=e}\}$	constructor
CE	$::=$	class expression	μ	$::= mod \ C \ M \ (\overline{C \ x})\{\text{return } e;\}$	method
	B	basic class		$\mid \text{abstract } C \ M \ (\overline{C \ x});$	method
	$\mid C$	class name	mod	$::= \text{abstract} \mid \text{virtual}$	member modifier
	$\mid \text{merge } CE_1, CE_2$	merge		$\mid \text{frozen} \mid \text{local}$	member modifier
	$\mid CE_1 \text{ override } CE_2$	override	e	$::=$	expression
	$\mid \text{rename } N \text{ to } N' \text{ in } CE$	rename		x	variable
	$\mid \text{restrict } N \text{ in } CE$	restrict		$\mid e.F$	client access
	$\mid \text{hide } N \text{ in } CE$	hide		$\mid e.M(\overline{e})$	client invocation
	$\mid \dots$			$\mid F$	internal access
		primitive operators		$\mid M(\overline{e})$	internal invocation
	$\mid CE_1 + CE_2$	sum		$\mid \text{new } C(\overline{e})$	object creation
	$\mid \text{rename } \sigma^t; \sigma^o \text{ in } CE$	reduct			
	$\mid \text{freeze } N \text{ in } CE$	freeze			
	$\mid CE[kh\{\text{super}(\overline{e})\}]$	constructor wrapper			
σ	$::= \overline{N:T \mapsto N':T', _ \mapsto N:T}$	renaming			
N	$::= F \mid M$	member name			
T	$::= C \mid MT$	member type			
MT	$::= \overline{C} \rightarrow C$	method type			
kh	$::= \text{constructor}(\overline{C \ x})$	constructor header			

Figure 1. FJIG (surface) syntax

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 initializations in both constructors are performed. Hence, an instance of class C1 has five fields. An invocation `new C1(5).M()` will return 11, since F3 in the body of M refers to the field declared in A1. 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. Expressions (method bodies) are similar to those of FJ (casts are omitted for brevity). However, 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 $e.M(\dots)$ behaves differently from $M(\dots)$ even in the case e denotes an object of the same class (that is, local members *do not* correspond to private members 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. Besides the operators mentioned above, other operators of the Jigsaw framework are *restrict*, which eliminates the definition for a

member, and *hide*, which makes a member no longer accessible from the outside. We refer to [9] and [3] for more details. All these operators and many others can be encoded by using a minimal set of *primitive* operators: *sum*, *reduct*, *freeze*, and *constructor wrapping*. The first three have already been proved in [3] to subsume the Jigsaw operators. To support the instantiation on a Java-like language, we also provide a *wrapping* operator to change the constructor of a class. The four operators are explained in the next section, when we present the semantics of FJIG.

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

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

Suppose we want to specialize the class `DBSerializer` for the DB server MySQL. We can create this specialization, called `MySQLSerializer`, in two steps: first, we provide an implementation of method `openConnection` with the specific code for MySQL, then we *hide* it, since clients of `MySQLSerializer` should never invoke this method directly.

We start by defining an auxiliary class `_MySQLSerializer`, merging `DBSerializer` with an anonymous basic class:

```
class _MySQLSerializer
```

```

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

```

Note the use of the constructor wrapper: the constructor of the anonymous basic class has a `String` parameter, whereas that of the class `DBSerializer`, which has no fields, is the default constructor. Hence, a constructor wrapper is inserted, which just invokes the default constructor of `DBSerializer`, so that the constructors of arguments of `merge` have the same parameters, allowing an invocation like `new _MySQLSerializer("mystring")`. As mentioned before, the class `_MySQLSerializer` provides, along the method `execute`, the method `openConnection` that we can hide as follows:

```

class MySQLSerializer
  hide openConnection in _MySQLSerializer

```

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 */
  }
}

```

Using this class with `MySQLSerializer` is not a problem, even though the inherited method `DBSerializer.execute` writes the data by invoking the method `serialize` and not `write`, since we can rename the method before merging the two classes:

```

class MySQLPersonSerializer
  hide serialize in
  override
  (rename write to serialize in Person) [
    constructor(String cs){super()}
  ],
  MySQLSerializer

```

2. FJIG calculus

The syntax of the calculus is given in Figure 2. Besides class names, (external) names and variables, we assume an infinite set of *internal (member) names* n .

Except for some shorter keywords for saving space, the only differences w.r.t. the surface syntax given in Figure 1 (restricted to the four primitive operators) are the following:

- There are no modifiers, since their semantics is encoded by distinguishing between *external* and *internal* member names, as explained in detail below. This solution is typical of module calculi [19, 3], and allows a simpler and intuitive model of composition operators. Internal names are used to refer to class members inside code (method bodies), and can be safely α -renamed. On the contrary, external names are used in class composition via operators and in selection of class 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.
- Expressions include *runtime expressions*, that is, (pre-)objects and blocks.

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 in 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 [14]. 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 denote by *dom* and *cod* the domain and codomain of a map, respectively. Given a basic class $[l \mid o \mid \rho]$, with $\rho = \{\bar{\varphi} \kappa \bar{\mu}\}$, we denote by $dom(\bar{\mu})$ and $dom(\bar{\varphi})$ the sets of internal names declared in $\bar{\mu}$ and $\bar{\varphi}$, 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 $N \in cod(\iota) \setminus dom(o)$, *virtual* if $N \in cod(\iota) \cap dom(o)$, *frozen* if $N \in dom(o) \setminus cod(\iota)$. 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, $cod(o) \subseteq dom(\rho)$, 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 in method bodies are translated by N , whereas for internal references the translation depends on the member kind. That is:

- 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 ,
- 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' .

In initialization expressions in constructor bodies, a field name F on the left-hand side is always translated by f' .

For instance, class `C` of previous section is translated by

$$\begin{aligned}
& [m_2:() \rightarrow \text{int} \mapsto M_2 \mid M_1:() \rightarrow \text{int} \mapsto m'_1, M_2:() \rightarrow \text{int} \mapsto m'_2, \mid \rho] \\
& \rho = \{ \\
& \quad \kappa() \{ \} \\
& \quad \text{int } m'_1() \{ \text{return } 1 + m_2; \} \\
& \quad \text{int } m'_2() \{ \text{return } m'_1 + m'_3; \} \\
& \quad \text{int } m'_3() \{ \text{return } 1; \} \\
& \}
\end{aligned}$$

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 there is an initialization expression. Note the difference with the form `new C(\bar{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

p	$::= \overline{cd}$	
cd	$::= C \mapsto CE$	
CE	$::= B \mid C \mid CE_1 + CE_2 \mid \sigma^\iota CE \mid_{\sigma^\circ} \mid freeze_N CE \mid CE[\overline{K(C\ x)}\{\bar{e}\}]$	
σ	$::= \overline{N:T \mapsto N':T'}, _ \mapsto N:T$	
N	$::= F \mid M$	
T	$::= C \mid MT$	
MT	$::= \overline{C \rightarrow C}$	
B	$::= [\iota \mid o \mid \rho]$	
ι	$::= \overline{n: T \mapsto N}$	input map
o	$::= \overline{N: T \mapsto n}$	output map
n	$::= f \mid m$	internal member name
ρ	$::= \{\overline{\varphi \ \kappa \ \bar{\mu}}\}$	local part
φ	$::= C \ f;$	
κ	$::= \overline{K(C\ x)\{f=e\}}$	
μ	$::= C \ m(C\ x)\{\mathbf{return} \ e;\}$	
e	$::= x \mid e.F \mid e.M(\bar{e}) \mid f \mid m(\bar{e}) \mid \mathbf{new} \ C(\bar{e})$	
	$\mid [\bar{\mu}; v \mid e]$	block
	$\mid C(f=e)$	(pre-)object
v, v^C	$::= C(\overline{f=e})$	value (object)

Figure 2. Syntax

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 $[\bar{\mu}; v \mid e]$, which model the execution of e where method internal names are bound in $\bar{\mu}$ and field internal names in the current object v . Hence, denoting by $dom(v)$ the set $\{f_1, \dots, f_n\}$ if $v = C(f_1 = v_1 \dots f_n = e_n)$, the block expression is well-formed only if $names(e) \subseteq dom(\bar{\mu}) \cup dom(v)$ 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 always be 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 Figure 3.

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 subterms of class expressions.

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 (analogously for other maps). Hence, rule (SUM) can only be applied (implicit side conditions) when the two sets of local names are disjoint ($dom(\rho_1) \cap dom(\rho_2) = \emptyset$), as are the sets of output names ($dom(o_1) \cap dom(o_2) = \emptyset$). The former condition can always be 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). Input names are

required to be the same, and the two constructors are also required to have the same parameters. This is not restrictive since these components can be always made equal by reduct and constructor wrapping operators, respectively.

In rule (REDUCT) the symbol \circ denotes composition of maps. New input and output names are chosen, modeled by $cod(\sigma^\iota)$ and $dom(\sigma^\circ)$, 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° . Input names can be shared or added, whereas output names can be duplicated or removed. Composition is well-formed only if type annotations are the same and the annotation of the new name is kept in the resulting map. That is: if ι contains $n: T \mapsto N$, then σ^ι should contain $N: T \mapsto N': T'$, and $\sigma^\iota \circ \iota$ will contain $n: T \mapsto N'$; if σ° contains $N': T' \mapsto N: T$, then o should contain $N: T \mapsto n$, and $o \circ \sigma^\circ$ will contain $N': T' \mapsto n$.

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.

Rule for constructor wrapping just correspond to provide a new constructor for a class.

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

The first rule is 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 access 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 field access or method invocation 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.⁴ Then, the following two rules can be applied.

⁴ 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.

$$\begin{array}{c}
\text{(CDEC1)} \frac{CE \longrightarrow CE'}{p, C \mapsto CE \longrightarrow p, C \mapsto CE'} \\
\text{(CDEC2)} \frac{}{p, C \mapsto B \longrightarrow p[B/C], C \mapsto B} \\
\text{(SUM)} \frac{[\iota | o_1 | \rho_1] + [\iota | o_2 | \rho_2] \longrightarrow [\iota | o_1, o_2 | \rho]}{\rho_i = \{\overline{\varphi}_i \text{K}(C \ x) \{f=e_i\} \overline{\mu}_i\}, i \in \{1, 2\} \\ \rho = \{\overline{\varphi}_1, \overline{\varphi}_2 \text{K}(C \ x) \{f=e_1, f=e_2\} \overline{\mu}_1, \overline{\mu}_2\}} \\
\text{(REDUCT)} \frac{}{\sigma^\iota [\iota | o | \rho] |_{\sigma^\circ} \longrightarrow [\sigma^\iota \circ \iota | o \circ \sigma^\circ | \rho]} \\
\text{(FREEZE)} \frac{\text{freeze}_N[\iota, n_1: T \mapsto N \dots n_k: T \mapsto N | o | \rho] \longrightarrow [\iota | o | \rho[n'/n_1] \dots [n'/n_k]]}{n' = o(N) \\ N \notin \text{cod}(\iota)} \\
\text{(K WRAPPING)} \frac{[\iota | o | \{\overline{\varphi} \text{K}(C_1 \ x_1 \dots C_n \ x_n) \{f=e\} \overline{\mu}\}] [\text{K}(C \ x) \{\overline{e}\}] \longrightarrow \overline{x} = x_1 \dots x_n}{[\iota | o | \{\overline{\varphi} \text{K}(C \ x) \{f=e[\overline{e}/\overline{x}]\} \overline{\mu}\}]}
\end{array}$$

$$\begin{array}{c}
\text{(CTX)} \frac{e \longrightarrow_p e'}{\mathcal{E}\{e\} \longrightarrow_p \mathcal{E}\{e'\}} \quad \text{(CLIENT-FIELD)} \frac{v^C \cdot F \longrightarrow_p [\overline{\mu}; v^C | f]}{p(C) = [\iota | o | \{\overline{\varphi} \ \kappa \ \overline{\mu}\}] \\ o(F) = f} \\
\text{(CLIENT-INVK)} \frac{v^C \cdot M(\overline{v}) \longrightarrow_p [\overline{\mu}; v^C | m(\overline{v})]}{p(C) = [\iota | o | \{\overline{\varphi} \ \kappa \ \overline{\mu}\}] \\ o(M) = m} \\
\text{(INT-FIELD)} \frac{[\overline{\mu}; v | \mathcal{E}\{f\}] \longrightarrow_p [\overline{\mu}; v | \mathcal{E}\{v_i\}]}{f \notin HB(\mathcal{E}) \\ v = C(f_1 = v_1 \dots f_n = v_n) \\ f = f_i} \\
\text{(INT-INVK)} \frac{[\overline{\mu}; v^C | \mathcal{E}\{m(\overline{v})\}] \longrightarrow_p [\overline{\mu}; v^C | \mathcal{E}\{e[\overline{v}/\overline{x}][v^C/\text{this}]\}]}{m \notin HB(\mathcal{E}) \\ \overline{\mu}(m) = (C_1 \ x_1 \dots C_n \ x_n) \{\text{return } e;\} \\ \overline{x} = x_1 \dots x_n} \\
\text{(OBJ-CREATION)} \frac{\text{new } C(\overline{v}) \longrightarrow_p C(f=e[\overline{v}/\overline{x}])}{p(C) = [\emptyset | o | \rho] \\ \rho = \{\overline{\varphi} \text{K}(C_1 \ x_1 \dots C_n \ x_n) \{f=e\} \overline{\mu}\} \\ \overline{x} = x_1 \dots x_n} \\
\text{(EXIT-BLOCK)} \frac{[\overline{\mu}; v | e] \longrightarrow_p e}{\text{names}(e) = \emptyset}
\end{array}$$

Figure 3. Flattening semantics

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 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. 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' . We omit the standard formal definition of hole binders $HB(\mathcal{E})$ of a context \mathcal{E} . Analogously, an internal method invocation is replaced by the corresponding body, found in the receiver's class, where parameters are replaced by arguments and **this** by the current object. Note that there are two kinds of references to the current object in a method body: through the keyword **this** (in client references, 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 field access/method invocation 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 access. 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.

3. 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 possibly retrieved from other classes and modified as effect of the module operators.

In order to define direct semantics, block expressions are generalized as shown in the top section of Figure 4. That is, besides the previous components, a block contains a *path map* 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 $\text{freeze}_N CE$, 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. A generalized block expression $[\hat{\iota}; \bar{\mu}; v \mid e]$ is well-formed only if $\text{names}(e) \subseteq \text{dom}(\hat{\iota}) \cup \text{dom}(\bar{\mu}) \cup \text{dom}(v)$ and these three sets are disjoint.

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 Figure 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, n is replaced by the local name n' found by lookup; moreover, the corresponding path map and 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 possibly retrieving and 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 Figure 3.

The remaining rule is analogous to that given for the flattening case. 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 $\text{lookup}_p(\hat{N}, \Lambda, C)$ is abbreviated by $\text{lookup}_p(\hat{N}, C)$.

The lookup function returns a triple consisting of input map, path map, methods and a local name, written $[\iota; \hat{\iota}; \bar{\mu} \mid n]$. However, the final result of lookup (that is, the result returned for the initial

call) is expected to be always of form $[\emptyset; \hat{\iota}; \bar{\mu} \mid n]$, abbreviated by $[\hat{\iota}; \bar{\mu} \mid n]$, 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 local 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 local 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 (we assume integer values and operations to make more readable examples):

$$\begin{aligned} C &\mapsto C_1 + C_2 \\ C_1 &\mapsto [\emptyset \mid \dots \mid \{\text{int } f; \text{K}()\{f = 3\} \dots\}] \\ C_2 &\mapsto [\emptyset \mid \dots, M \mapsto m \mid \\ &\quad \{\text{int } f; \text{K}()\{f = 5\} \text{int } m()\{\text{return } f + 1;\}\}] \end{aligned}$$

and the expression $\text{new } C().M()$.

An instance of class C has two fields, inherited from C_1 and C_2 , 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 lookup from C , and the search propagated to both C_1 and C_2 . Only lookup in C_2 is successful and returns the result

$$[\;; \text{int } m()\{\text{return } f + 1;\} \mid m]$$

which is modified in $[\;; \text{int } m()\{\text{return } f.2 + 1;\} \mid m]$ to take into account that the method has been found in the second argument. Hence, method invocation reduces to

$$[\;; \text{int } m()\{\text{return } f.2 + 1;\}; C(f.1 \mapsto 3, f.2 \mapsto 5) \mid m]$$

where the body of m correctly refers to the second field.

In flattening semantics, C reduces to the following basic class:

$$\begin{aligned} &[\emptyset \mid \dots, M \mapsto m \mid \rho] \\ \rho &= \{\text{int } f.1; \text{int } f.2; \kappa \text{int } m()\{\text{return } f.2 + 1;\} \dots\} \\ \kappa &= \text{K}()\{f.1 = 3, f.2 = 5\} \end{aligned}$$

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° . 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

π	$::= i_1 \dots i_k$	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	$::= \dots \mid [\hat{\iota}; \bar{\mu}; v \mid e]$	(generalized) block

$\text{(CTX)} \frac{e \rightarrow_p e'}{\mathcal{E}\{e\} \rightarrow_p \mathcal{E}\{e'\}}$	$\text{(CLIENT-INVK)} \frac{}{v^C.M(\bar{v}) \rightarrow_p [\hat{\iota}; \bar{\mu}; v^C \mid m(\bar{v})]} \quad \text{lookup}_p(M, C) = [\hat{\iota}; \bar{\mu} \mid m]$
$\text{(INT-FIELD)} \frac{}{[\hat{\iota}; \bar{\mu}; v \mid \mathcal{E}\{f\}] \rightarrow_p [\hat{\iota}; \bar{\mu}; v \mid \mathcal{E}\{v_i\}]}$	$\begin{array}{l} f \notin HB(\mathcal{E}) \\ v = C(f_1 = v_1 \dots f_n = v_n) \\ f = f_i \end{array}$
$\text{(INT-INVK)} \frac{}{[\hat{\iota}; \bar{\mu}; v \mid \mathcal{E}\{m(\bar{v})\}] \rightarrow_p [\hat{\iota}; \bar{\mu}; v \mid \mathcal{E}\{e[\bar{v}/\bar{x}][v^C/\text{this}]\}]}$	$\begin{array}{l} m \notin HB(\mathcal{E}) \\ \bar{\mu}(m) = C m(C_1 x_1 \dots C_n x_n)\{\text{return } e;\} \\ \bar{x} = x_1 \dots x_n \end{array}$
$\text{(PATH)} \frac{}{[\hat{\iota}, n \mapsto \pi; \bar{\mu}; v^C \mid e] \rightarrow_p [\hat{\iota}, \hat{\iota}'; \bar{\mu}[n'/n], \bar{\mu}'; v^C \mid e[n'/n]]}$	$\begin{array}{l} n \in \text{names}(e) \\ \text{lookup}_p(\pi, C) = [\hat{\iota}'; \bar{\mu}' \mid n'] \end{array}$
$\text{(OBJ-CREATION)} \frac{}{\text{new } C(\bar{v}) \rightarrow_p C(f = e[\bar{v}/\bar{x}])}$	$\begin{array}{l} k\text{-lookup}_p(C) = K(C_1 x_1 \dots C_n x_n)\{\overline{f=e}\} \\ \bar{x} = x_1 \dots x_n \end{array}$
$\text{(EXIT-BLOCK)} \frac{}{[\hat{\iota}; \bar{\mu}; v \mid e] \rightarrow_p e}$	$\text{names}(e) = \emptyset$

$\begin{array}{l} \text{lookup}_p(\hat{N}, \pi, C) = \text{lookup}_p(\hat{N}, \pi, CE) \\ \text{lookup}_p(N, \pi, [l \mid o, N \mapsto n \mid \{\bar{\varphi} \ \bar{\kappa} \ \bar{\mu}\}]) = [l; \emptyset; \bar{\mu} \mid n] \\ \text{lookup}_p(\hat{N}, \pi, CE_1 + CE_2) = \alpha_i([l; \hat{\iota}; \bar{\mu} \mid n]) \\ \text{lookup}_p(\hat{N}, \pi, \sigma^\iota CE _{\sigma^\circ}) = [\sigma^\iota \circ l; \hat{\iota}; \bar{\mu} \mid n] \end{array}$	$\begin{array}{l} \text{if } p(C) = CE \\ \text{if } \text{lookup}_p(\hat{N}, \pi.i, CE_i) = [l; \hat{\iota}; \bar{\mu} \mid n], i \in \{1, 2\} \\ \text{if } \text{lookup}_p(\hat{N}', \pi.1, CE) = [l; \hat{\iota}; \bar{\mu} \mid n], \\ \hat{N}' = \sigma^o(N) \text{ if } \hat{N} = N, \hat{N}' = \hat{N} \text{ otherwise} \\ \text{if } \hat{N} \neq \pi, \text{lookup}_p(\hat{N}, \pi.1, CE) = [l, n_1 \mapsto N \dots n_k \mapsto N; \hat{\iota}; \bar{\mu} \mid n], \\ N \notin \text{cod}(l) \\ \text{if } \text{lookup}_p(N, \pi.1, CE) = [l, n_1 \mapsto N \dots n_k \mapsto N; \hat{\iota}; \bar{\mu} \mid n], N \notin \text{cod}(l) \end{array}$
$\begin{array}{l} \text{lookup}_p(\hat{N}, \pi, \text{freeze}_N CE) = [l; \hat{\iota}, n_1 \mapsto \pi \dots n_k \mapsto \pi; \bar{\mu} \mid n] \\ \text{lookup}_p(\pi, \pi, \text{freeze}_N CE) = [l; \hat{\iota}, n_1 \mapsto \pi \dots n_k \mapsto \pi; \bar{\mu} \mid n] \\ \text{lookup}_p(\hat{N}, \pi, CE[K(\bar{C} \ \bar{x})\{\bar{e}\}]) = \text{lookup}_p(\hat{N}, \pi.1, CE) \end{array}$	$\begin{array}{l} \text{if } p(C) = CE \\ \text{if } k\text{-lookup}_p(C) = k\text{-lookup}_p(CE) \\ k\text{-lookup}_p([\emptyset \mid o \mid \{\bar{\varphi} \ \bar{\kappa} \ \bar{\mu}\}]) = \bar{\kappa} \\ k\text{-lookup}_p(CE_1 + CE_2) = K(\bar{C} \ \bar{x})\{\alpha_1(\overline{f=e}), \alpha_2(\overline{f'=e'})\} \\ k\text{-lookup}_p(\sigma^\iota CE _{\sigma^\circ}) = k\text{-lookup}_p(CE) \\ k\text{-lookup}_p(\text{freeze}_N CE) = k\text{-lookup}_p(CE) \end{array}$
$k\text{-lookup}_p(CE[K(\bar{C} \ \bar{x})\{\bar{e}\}]) = K(\bar{C} \ \bar{x})\{\overline{f=e[\bar{e}/\bar{x}]}\}$	$\text{if } k\text{-lookup}_p(CE) = K(C_1 x_1 \dots C_n x_n)\{\overline{f=e}\}, \bar{x} = x_1 \dots x_n$

Figure 4. Direct semantics

$$\begin{array}{l} C \mapsto_{M_1 \mapsto M'_1} C|_{M \mapsto M'} \\ C' \mapsto [m' \mapsto M_1 \mid M' \mapsto m \mid \{\dots \text{int } m() \{\text{return } m'();\}\}] \end{array}$$

and assume that some method invocation triggers lookup for M in C . Then, lookup is propagated under the name M' to C' . Lookup of M' in C' is successful and returns the result

$$[m' \mapsto M_1; ; \text{int } m() \{\text{return } m'();\} \mid m]$$

which is modified in

$$[m' \mapsto M'_1; ; \text{int } m() \{\text{return } m'();\} \mid m]$$

as an effect of the input renaming.

In flattening semantics, C reduces to the following basic class:

$$[m' \mapsto M'_1 \mid M \mapsto m \mid \{\dots \text{int } m() \{\text{return } m'();\}\}]$$

There are two clauses defining lookup on a freeze expression. The former handles most cases, except the special situation in which we are looking exactly for the member which has been frozen in the current subterm π which has form $\text{freeze}_N CE$. In this special case (second clause) 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), lookup is propagated, and the result of lookup on the argument is modified as in the previous case.

The following example illustrates the second clause. Consider the program

$$\begin{array}{l} C \mapsto \text{freeze}_p C' \\ C' \mapsto [f \mapsto F \mid F \mapsto f', M \mapsto m \mid \\ \{\text{int } f'; K()\{f' = 42\} \text{int } m() \{\text{return } f + 1;\}\}] \end{array}$$

and the expression $\text{new } C().M()$.

An instance of class C has one field, inherited from C' and initialized to 42. The above expression reduces to $C(f' \mapsto 42).M()$. Now, M is invoked, starting lookup from C , and the search propagated to C' . Lookup in C' is successful and returns the result

$$[f \mapsto F; \text{int } m() \{ \text{return } f + 1; \} | m]$$

which is modified in $[; f \mapsto \Lambda; \text{int } m() \{ \text{return } f' + 1; \} | m]$, where Λ denotes the empty path, to take into account that F has been frozen. Hence, method invocation reduces to

$$[f \mapsto \Lambda; \text{int } m() \{ \text{return } f + 1; \}; C(f \mapsto 42) | m]$$

where the body of m correctly refers to F frozen in the top level freeze.

$$\begin{aligned} & [\emptyset | F \mapsto f', M \mapsto m | \\ & \{ \text{int } f'; \text{K}() \{ f' = 42 \} \text{int } m() \{ \text{return } f' + 1; \} \} \end{aligned}$$

Figure 5 shows a more involved example which compares 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 the frozen method M whose body invokes the abstract method M' . Class B has one local field f initialized to 0 and defines the frozen method M' which returns this field. Class C is obtained by summing A and B and 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' which returns 8. 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 $\text{new } D().M()$ 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 $\xrightarrow{*}$ the reflexive and transitive closure of the flattening relation, and analogously for the reduction relation.

Theorem 1. *If $p \xrightarrow{*} p'$, then $e \xrightarrow{*}_p v$ iff $e \xrightarrow{*}_{p'} v$.*

To prove the theorem, we first of all define two congruence relations \sim and $\sim_{p,C}$ on lookup results, the latter indexed on programs and class names:

- \sim is the least congruence relation s.t.

$$[\iota; \hat{\iota}; \bar{\mu}, \mu | n] \sim [\iota; \hat{\iota}; \bar{\mu} | n]$$

$$\text{if } \mu = C \ m(\overline{C \ x}) \{ \text{return } e; \}, m \neq n, m \notin \text{names}(\bar{\mu}).$$

- $\sim_{p,C}$ is the least congruence relation including \sim and, moreover, s.t.

$$[\iota; \hat{\iota}, n \mapsto \pi; \bar{\mu} | n] \sim_{p,C} [\iota; \hat{\iota}, \hat{\iota}'; \bar{\mu}[n'/n], \bar{\mu}' | n']$$

$$\text{if } \text{lookup}_p(C, \pi) = [\hat{\iota}'; \bar{\mu}' | n'].$$

The former congruence states that a lookup result is equivalent to another where a useless method has been removed. The latter congruence states that a lookup result is equivalent to another where an association from internal name to path has been resolved in turn by lookup, and path map and methods have been expanded. Then, the proof is based on the following lemma.

Lemma 2. *If CE is the π -subterm of $p(C)$, and $CE \longrightarrow CE'$, then:*

- $\text{lookup}_p(N, \pi, CE) = [\iota; \hat{\iota}; \bar{\mu} | n]$ implies $\text{lookup}_p(N, \pi, CE') = [\hat{\iota}'; \hat{\iota}'; \bar{\mu}' | n]$ and $[\iota; \hat{\iota}; \bar{\mu} | n] \sim_{p,C} [\hat{\iota}'; \hat{\iota}'; \bar{\mu}' | n']$,
- $\text{lookup}_p(N, \pi, CE') = [\hat{\iota}'; \hat{\iota}'; \bar{\mu}' | n]$ implies $\text{lookup}_p(N, \pi, CE) = [\iota; \hat{\iota}; \bar{\mu} | n]$ and $[\iota; \hat{\iota}; \bar{\mu} | n] \sim_{p,C} [\hat{\iota}'; \hat{\iota}'; \bar{\mu}' | n']$.

Proof:

By induction on the definition of $CE \longrightarrow CE'$.

(SUM) We have

$$CE = CE_1 + CE_2$$

$$CE_1 = [\iota | o_1 | \{ \bar{\varphi}_1 \text{K}(\overline{C \ x}) \{ f_1 = e_1 \} \bar{\mu}_1 \} | n]$$

$$CE_2 = [\iota | o_2 | \{ \bar{\varphi}_2 \text{K}(\overline{C \ x}) \{ f_2 = e_2 \} \bar{\mu}_2 \} | n]$$

$$CE' = [\iota | o_1, o_2 | \{ \bar{\varphi}_1, \bar{\varphi}_2 \text{K}(\overline{C \ x}) \{ f_1 = e_1, f_2 = e_2 \} \bar{\mu}_1, \bar{\mu}_2 \} | n]$$

Moreover, $\text{lookup}_p(N, \pi, CE)$ and $\text{lookup}_p(N, \pi, CE')$ are defined only if $(o_1, o_2)(N) = n$ for some n . By well-formedness of o_1, o_2 this means that either $o_1(N)$ is defined or $o_2(N)$ is defined, but not both. Let us assume $o_1(N) = n$ (the other case is analogous). Then,

$$\text{lookup}_p(N, \pi, CE) = [\iota; \emptyset; \alpha_1(\bar{\mu}_1) | n]$$

$$\text{lookup}_p(N, \pi, CE') = [\iota; \emptyset; \alpha_1(\bar{\mu}_1), \alpha_2(\bar{\mu}_2) | n]$$

and these two lookup results are \sim -equivalent since, by well-formedness of CE' , methods in $\alpha_2(\bar{\mu}_2)$ are useless (formally, $(\{n\} \cup \text{names}(\alpha_1(\bar{\mu}_1))) \cap \text{dom}(\alpha_2(\bar{\mu}_2)) = \emptyset$).

(REDUCT) We have

$$CE = \sigma^\iota[\iota | o | \{ \bar{\varphi} \text{K} \bar{\mu} \} | \sigma^o]$$

$$CE' = [\sigma^\iota \circ \iota | o \circ \sigma^o | \{ \bar{\varphi} \text{K} \bar{\mu} \} | n]$$

Moreover, $\text{lookup}_p(N, \pi, CE)$ and $\text{lookup}_p(N, \pi, CE')$ are defined only if $o(N) = n$ for some n . Then,

$$\text{lookup}_p(N, \pi, CE) = [\sigma^\iota \circ \iota; \emptyset; \bar{\mu} | n]$$

$$\text{lookup}_p(N, \pi, CE') = [\sigma^\iota \circ \iota; \emptyset; \bar{\mu} | n]$$

and we get the thesis.

(FREEZE) We have

$$CE = \text{freeze}_N[\iota, n_1: T \mapsto N' \dots n_k: T \mapsto N' | o | \{ \bar{\varphi} \text{K} \bar{\mu} \} | n]$$

$$CE' = [\iota | o | \{ \bar{\varphi} \text{K} \bar{\mu}[n'/n_1] \dots [n'/n_k] \} | n]$$

$$n' = o(N')$$

$$N \notin \text{cod}(\iota)$$

Moreover, $\text{lookup}_p(N, \pi, CE)$ and $\text{lookup}_p(N, \pi, CE')$ are defined only if $o(N) = n$ for some n . Then,

$$\text{lookup}_p(N, \pi, CE) = [\iota; n_1: T \mapsto \pi \dots n_k: T \mapsto \pi; \bar{\mu} | n]$$

$$\text{lookup}_p(N, \pi, CE') = [\iota; \emptyset; \bar{\mu}[n'/n_1] \dots [n'/n_k] | n]$$

Since CE is the π -subterm of $p(C)$, $\text{lookup}_p(\pi, \Lambda, C) =$

$$\text{lookup}_p(\pi, \pi, CE) =$$

$$[\iota; n_1: T \mapsto \pi \dots n_k: T \mapsto \pi; \bar{\mu}[n'/n_1] \dots [n'/n_k] | n'], \text{ hence the thesis follows.}$$

(WRAPPING) Trivial.

(CTX) The proof is by structural induction on the context.

4. Conclusion

We have defined flattening and direct semantics, and proved their equivalence, for FJIG, a class-based calculus formalizing the Jigsaw framework, originally proposed in Bracha's seminal work [9], in a Java-like setting.

There are usually two intuitive models to understand inheritance: one where inherited methods are copied into heir classes and one where member lookup is performed by ascending the inheritance chain. This paper shows the equivalence of these two views in a

$$\begin{aligned}
v^D &\equiv D(f.2.1 = 0) \\
\mu &\equiv C m() \{\mathbf{return} m'();\} \\
\mu'' &\equiv C m''() \{\mathbf{return} f.2.1;\} \\
\bar{\mu}_D &\equiv C m() \{\mathbf{return} m''();\}; C m''() \{\mathbf{return} f.2.1;\}; C m'''() \{\mathbf{return} 8;\}
\end{aligned}$$

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

$$\begin{aligned}
&\mathit{freeze}_{M'}(A + B) \longrightarrow \\
&\mathit{freeze}_M[m' \mapsto M' \mid M \mapsto m, M' \mapsto m'' \mid \{C f.2; \mathbf{K}\}(f.2 = 0) C m() \{\mathbf{return} m'();\}; C m''() \{\mathbf{return} f.2;\}] \longrightarrow \\
&[\emptyset \mid M \mapsto m, M' \mapsto m'' \mid \{C f.2; \mathbf{K}\}(f.2 = 0) C m() \{\mathbf{return} m''();\}; C m''() \{\mathbf{return} f.2;\}]
\end{aligned}$$

$$\begin{aligned}
&\emptyset \mid C_{_ \mapsto M', M \mapsto M} + [\emptyset \mid M' \mapsto m' \mid \{\mathbf{K}()\} C m'() \{\mathbf{return} 8;\}] \longrightarrow \\
&[\emptyset \mid M \mapsto m \mid \{C f.2; \mathbf{K}\}(f.2 = 0) C m() \{\mathbf{return} m''();\}; C m''() \{\mathbf{return} f.2;\}] \\
&+ [\emptyset \mid M' \mapsto m' \mid \{\mathbf{K}()\} C m'() \{\mathbf{return} 8;\}] \longrightarrow \\
&[\emptyset \mid M \mapsto m, M' \mapsto m''' \mid \{C f.2.1; \mathbf{K}\}(f.2.1 = 0) \bar{\mu}_D]
\end{aligned}$$

$$\begin{aligned}
p' &\equiv A = [m' \mapsto M' \mid M \mapsto m \mid \{\mathbf{K}()\} C m() \{\mathbf{return} m'();\}] \\
&\quad B = [\emptyset \mid M' \mapsto m' \mid \{C f; \mathbf{K}\}(f = 0) C m'() \{\mathbf{return} f;\}] \\
&\quad C = [\emptyset \mid M \mapsto m, M' \mapsto m'' \mid \{C f.2; \mathbf{K}\}(f.2 = 0) C m() \{\mathbf{return} m''();\}; C m''() \{\mathbf{return} f.2;\}] \\
&\quad D = [\emptyset \mid M \mapsto m, M' \mapsto m''' \mid \{C f.2.1; \mathbf{K}\}(f.2.1 = 0) \bar{\mu}_D]
\end{aligned}$$

$$\mathbf{new} D().M() \longrightarrow_{p'} v^D.M() \longrightarrow_{p'} [\bar{\mu}_D; v^D \mid m()] \longrightarrow_{p'} [\bar{\mu}_D; v^D \mid m'()] \longrightarrow_{p'} [\bar{\mu}_D; v^D \mid f.2.1] \longrightarrow_{p'} [\bar{\mu}_D; v^D \mid 0] \longrightarrow_{p'} 0$$

$$\begin{array}{ll}
\mathbf{new} D().M() \longrightarrow_p & k\text{-lookup}_p(D) = \mathbf{K}\{f.2.1 = 0\} \\
v^D.M() \longrightarrow_p & \text{lookup}_p(M, \Lambda, D) = [\Lambda; m' \mapsto 1.1; \mu \mid m] \\
[m' \mapsto 1.1; \mu; v^D \mid m()] \longrightarrow_p & \\
[m' \mapsto 1.1; \mu; v^D \mid m'()] \longrightarrow_p & \text{lookup}_p(1.1, \Lambda, D) = [\Lambda; \Lambda; \mu'' \mid m''] \\
[m' \mapsto 1.1; \mu; \mu''; v^D \mid m''()] \longrightarrow_p & \\
[m' \mapsto 1.1; \mu; \mu''; v^D \mid f.2.1] \longrightarrow_p & \\
[m' \mapsto 1.1; \mu; \mu''; v^D \mid 0] \longrightarrow_p & \\
0 &
\end{array}$$

Figure 5. Example

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.

As already mentioned in the Introduction, many proposals for extending the object-oriented paradigm have just taken one approach or the other. In particular, the most direct source of inspiration for our work has been [16], 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.

On the other hand, to the best of our knowledge there has been no attempt at providing both semantics and proving their equivalence, as we do in this paper, for any of these extensions.

One interesting direction of further work can be to investigate whether and how the equivalence can be preserved in a language allowing features whose runtime behaviour depends on static types, such as overloading or static binding of members. We also plan to

investigate smart implementation techniques of the direct semantics in a prototype interpreter we are developing.

Acknowledgments We warmly thank the anonymous referees for many useful comments.

References

- [1] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam: A smooth extension of Java with mixins. In E. Bertino, editor, *ECOOP'00 - European Conference on Object-Oriented Programming*, number 1850 in Lecture Notes in Computer Science, pages 154–178. Springer, 2000. An extended version is [2].
- [2] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam—designing a Java extension with mixins. *ACM Transactions on Programming Languages and Systems*, 25(5):641–712, September 2003. Extended version of [1].
- [3] Davide Ancona and Elena Zucca. A calculus of module systems. *Journ. of Functional Programming*, 12(2):91–132, 2002.

- [4] Davide Ancona, Elena Zucca, and Sophia Drossopoulou. Overloading and inheritance. In *FOOL'01 - Intl. Workshop on Foundations of Object Oriented Languages*, January 2001.
- [5] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Comput. Lang. Syst. Struct.*, 34(2-3):83–108, 2008.
- [6] Lorenzo Bettini, Sara Capecchi, and Betti Venneri. Double Dispatch in C++. *Software - Practice and Experience*, 36(6):581 – 613, 2006.
- [7] Lorenzo Bettini, Sara Capecchi, and Betti Venneri. Featherweight Java with multi-methods. In *PPPJ'07 - Principles and Practice of Programming in Java*, volume 272, pages 83–92. ACM Press, 2007.
- [8] Lorenzo Bettini, Sara Capecchi, and Betti Venneri. Featherweight Java with dynamic and static overloading. *Science of Computer Programming*, 2008. To appear.
- [9] Gilad Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.
- [10] Kathleen Fisher and John Reppy. A typed calculus of traits. In *FOOL'04 - Intl. Workshop on Foundations of Object Oriented Languages*, 2004.
- [11] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *ACM Symp. on Principles of Programming Languages 1998*, pages 171–183. ACM Press, 1998.
- [12] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.
- [13] Atsushi Igarashi and Benjamin C. Pierce. On inner classes. *Information and Computation*, 177(1):56–89, August 2002.
- [14] Giovanni Lagorio, Marco Servetto, and Elena 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. Submitted for publication.
- [15] Luigi Liquori and Arnaud Spiwack. Extending FeatherTrait Java with interfaces. *Theoretical Computer Science*, 2008.
- [16] Luigi Liquori and Arnaud Spiwack. FeatherTrait: A modest extension of Featherweight Java. *ACM Transactions on Programming Languages and Systems*, 30(2), 2008.
- [17] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *ACM Symp. on Principles of Programming Languages 1997*. ACM Press, 1997.
- [18] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *ECOOP'03 - Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer, 2003.
- [19] J. B. Wells and R. Vestergaard. Confluent equational reasoning for linking with first-class primitive modules. In *ESOP 2000 - European Symposium on Programming 2000*, number 1782 in *Lecture Notes in Computer Science*, pages 412–428. Springer, 2000.