JOURNAL OF OBJECT TECHNOLOGY
Published by AITO — Association Internationale pour les Technologies Objets, ©JOT 2011
Online at http://www.jot.fm.

DEEPFJIG
Modular composition of nested classes

Andrea Corradi[a] Marco Servetto[b] Elena Zucca[a]

a. DIBRIS - University of Genova
Via Dodecaneso, 35
16146 Genova, Italy

b. School of Engineering and Computer Science
Victoria University of Wellington
PO Box 600
Wellington 6140, New Zealand

Abstract We present a new language design which smoothly integrates *modular composition* and *nesting* of Java-like classes. That is, inheritance has been replaced by an expressive set of composition operators, inspired by Bracha's Jigsaw framework, and these operators allow to manipulate (e.g., rename or duplicate) a nested class at any level of depth. Typing is *nominal* as characteristic of Java-like languages, so types are paths of the form $\texttt{outer}^n.C_1.\ \ldots\ .C_k$ which, depending on the class (node) where they occur, denote another node in the nesting tree. However, paths denoting the same class are *not* equivalent, since they behave differently w.r.t. composition operators.

The resulting language, called DEEPFJIG, obtains a great expressive power, allowing, e.g., to solve the expression problem, encode basic AOP mechanisms, and bring some refactoring techniques at the language level, while keeping a very simple semantics and type system which represent a natural extension for, say, a Java programmer.

Keywords Java, module composition, nested classes

Introduction

Featherweight Jigsaw [LSZ09a, LSZ09b, LSZ12] (FJIG for short) is a simple calculus where basic building blocks are classes in the style of Featherweight Java (FJ for short) [IPW99], but inheritance has been replaced by the much more flexible notion originally proposed in Bracha's Jigsaw framework [Bra92]. 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, composed by: *sum*, *restrict*, *alias* and *redirect*.

In this paper, we describe an extension of FJIG, called DEEPFJIG, where these composition operators have been generalized to manipulate nested classes. For instance, sum of two classes is hierarchical in the sense that nested classes with the same name are recursively summed, similarly to *deep mixin composition* [Ern99b, OZ05, Hut06] and *family polymorphism* [EOC06, IV07, ISV08], which, however, take an asymmetric approach. Analogously it is possible to rename or make an alias of a field, method, or a nested class itself, at any depth level.

Typing is nominal as characteristic of Java-like languages, that is, types are *(class) paths*, which are sequences of the form $\texttt{outer}^n.C_1.\ \ldots\ .C_k$ which, depending on the class (node) where they occur, denote another node in the nesting tree. However, class paths denoting the same class are *not* equivalent, since they behave differently w.r.t. composition operators.

The resulting language offers a great expressive power, allowing, e.g., to solve the expression problem and to encode generics [BOSW98, GJSB05] and **`MyType`** [BOW98]. Moreover, since a whole program can be "packed" into a single class, also the basic AOP mechanisms can be expressed. Finally, the kind of code manipulation achieved by the composition operators corresponds to bring some refactoring techniques at the linguistic level. On the other hand, the generalization of the composition operators to the case with nesting is very natural and intuitive, and, more generally, the language keeps a simple semantics and type system which represent a natural extension for, say, a Java programmer.

There are many other proposals allowing class nesting and, hence, some form of paths. Notably, among real world languages, Java and and C# support nested classes only as a way to achieve hierarchical organization. In such languages, as well as in Scala, the binding for nested classes is static, that is, redeclaring a nested class in a subclass has the effect of *hiding* the parent's nested class. On the other hand, in, e.g., gbeta [Ern99a] and Newspeak [BvdAB+10], as in the literature on family polymorphism [Ern01, EOC06, ISV05, IV07, ISV08], a class can inherit from a *virtual* superclass, and this feature provides a great expressive power, at the price of making typechecking harder. Virtual superclasses can be emulated by C++ templates, as shown in the work on mixin layers [SB01]. In Scala, abstract types and traits can be used for a similar, but more involved, emulation.

Our approach comes from a design principle rather different from all those mentioned above. That is, we replace inheritance by a true language of composition operators, mainly inspired by the seminal work in [Bra92] and its formalization as module calculus in [AZ02], and partly by the work on traits [SDNB03], notably the proposals which do not include inheritance [BDG07, BDG08]. In DEEPFJIG this design principle is "naturally" extended to handle nested modules (classes), and this enables us to use a type system without dependent types and without class families, but still powerful enough. This shows a particular trade-off in the language design space that has a number of useful properties, including better compatibility with the mainstream than proposals using dependent types. For what concerns our form of class paths, the closest work is likely [IV07], notably for the formalization aspects. A more detailed comparison with related work is provided in Section 4.

The rest of the paper is organized as follows. In Section 1 we illustrate DEEPFJIG and its expressive power. In Section 2 we give the formal syntax and semantics, and in Section 3 the type system and the related results. In Section 4 we summarize the contribution of the paper and discuss related work, and in Section 5 we conclude outlining some further research directions. Proofs of results are in the Appendix. This paper is an improved and extended version of [CSZ10, CSZ11]. Notably, the full formalization of the semantics, the type system and the soundness results were not included in [CSZ11].

1 Examples

We illustrate first, in Section 1.1, features which are inherited from FJIG, then we describe how to declare and refer to nested classes in Section 1.2, and in Section 1.3 the composition operators. Finally, in Section 1.4 we provide some more interesting examples which show the expressive power of the language.

1.1 FJIG summary

The following example shows three class declarations.

```
A = abstract{
  abstract int m1();
  int m2() { return this.m1() + 1; }
}
B = { int m1() { return 1; } }
C = A
```

The first two declarations look similar to Java class declarations. However, the syntax is slightly different, to stress that in FJIG and DEEPFJIG a class declaration just introduces a name for the expression occurring at the right of the equal symbol, which is called a *class expression* and denotes an unnamed class. See also Section 4 for more comments on this difference. In the first two declarations above, the class expression is a *basic class*, which is similar to a Java class body. In the third declaration, the class expression is the class name `A`, which denotes the first class, hence the declaration is equivalent, in a sense that will be made more precise in the last paragraph of this subsection, to the following one:

```
C = abstract{
  abstract int m1();
  int m2(){return this.m1() + 1;}
}
```

Compound class expressions can be constructed using *composition operators*[1]. For instance, a new class can be defined by applying the *sum* operator to those above as follows:

```
Sum = A [+] B
```

This declaration is equivalent to the following:

```
Sum = {
  int m1() { return 1; }
  int m2() { return this.m1() + 1; }
}
```

Conflicting definitions for the same field or method are not permitted, whereas **`abstract`** fields or methods with the same name are shared.

The modifier **`abstract`** applies to fields as well, as shown by the following example which also illustrates how constructors work. The class declarations

```
A1 = abstract{
  abstract int f1;
  int f2; constructor(int x) { this.f2 = x; }
  int m() { return this.f1 + this.f2; }
}
B1 = {
  int f1; constructor(int x) { this.f1 = x + 1; }
} [+] A1
```

are equivalent to

[1] Each basic class plays the role of a constant (0-ary) composition operator, see the formal syntax in Figure 1.

```
A1 = { /*as before*/ }
B1 = {
  int f1, f2;
  constructor(int x) { this.f1 = x + 1; this.f2 = x; }
  int m() { return this.f1 + this.f2; }
}
```

A basic class defines one 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 and empty body for classes having no defined fields. In order to be composed by the sum operator, two classes should provide a constructor with the same parameter list. The effect is that the resulting class provides a constructor with the same parameter list, that executes both of the original constructors.

In order to be able to sum two classes with different constructor headers, FJIG provides a *constructor wrapper* operator which allows the programmer to make them equal. In this paper, we have preferred not to generalize this approach to the nested case, since in any case it would not be adequate in a realistic language. Indeed, with a naive introduction of full Java constructors sum would no longer be symmetric, due to the presence of side effects, whereas symmetric composition is a key feature we want to keep. Replacing constructors with object creation expressions, as in Javascript, Emerald [RTL+91] or Grace [BBN10], could be an elegant solution, which we leave to further work.

Note that, analogously to abstract method declarations, abstract field declarations allow a class to use a field without initializing it. In this way, classes composed by sum can share the same field, provided it is defined in (at most) one. Note that this corresponds to *sharing* fields as in, e.g., [BDNW08]; however, in our framework we do not need an ad-hoc notion.

Flattening versus direct semantics Before introducing nested classes, let us briefly discuss the notion of "equivalence" we have used above to informally explain the semantics of DEEPFJIG. This equivalence will be formalized in the next section (see Figure 2) by a relation, called *flattening*, which translates DEEPFJIG into a "flat" language where class expressions are only basic classes (hence there are no longer composition operators). Analogously, the semantics of inheritance in object-oriented languages can be explained by saying that, roughly, the effect is the same one would get by duplicating the methods of the parent class in the heir. However, inheritance can also be explained in a different way, by describing a runtime procedure called *method look-up*. In other words, semantics of inheritance can be given either by translation into a language with no inheritance, or by a direct execution model. In this paper, we choose to explain the semantics of DEEPFJIG by flattening since this provides a more simple and intuitive understanding of the effect of the operators. However, a direct semantics could be provided for DEEPFJIG along the lines of that we have described for FJIG [LSZ09a, LSZ09b, LSZ12], even though method look-up is much more involved when there are many composition operators rather than just **extends**. Note also that an implementation could be based on one of these two approaches, or more likely adopt some even different or mixed optimized approach.

1.2 Nested classes

In DEEPFJIG, a basic class can also contain declarations of *nested classes*, as shown in the example below.

```
{
  A = { B = { ... }} [+]
    {
      B = {
        C = { <> m() { return new <>(); } }
        D = C [+] outer.outer.G
      }
```

```
    E = B.C [+] outer.G
    F = { ... }
  }
 G = { ... }
 H = A.B.C [+] G
}
```

Hence, a basic class has a tree shape, where the basic class is the root, the children of a basic class are its nested classes, and the children of a nested class are the basic classes appearing in its defining expression. For instance, the basic class in the example has three children, `A`, `G`, and `H`, and nested class `A` has two children, since it is defined as sum of two basic classes.

In the following, we will informally use "position" to designate a node in the tree which is either the root or a nested class. Note that, differently from absolute paths in other approaches supporting nested classes, a position in the tree cannot be identified by just a sequence of class names, due to the presence of class composition operators, see, for instance, the two nested classes named `B` which could be identified by, e.g., `A.1.B` and `A.2.B`.

Sequences of the form $\texttt{outer}^n.C_1.\ \ldots\ .C_k$, with $n, k \geq 0$, called *(class) paths*, denote a class, and can be used the same way as class names, that is, as types, in **`new`** expressions and as subterms of class expressions, as shown in the example. Class paths can be classified along two orthogonal dimensions:

- paths with $k = 0$ denote enclosing basic classes, whereas paths with $k > 0$ denote nested classes. In particular, the path Λ (written <> in code), that is, the unique path where $n, k = 0$, denotes the directly enclosing basic class. As the reader may have noted, this path has many analogies with the **`MyType`** notion in literature. However, we prefer the notation <> to stress that it is just a special case of path and that no sophisticated notion is needed in the type system, see more comments in Section 1.4.

- paths with $n = 0$, called *downward (class) paths*, refer to a class in the current scope (basic class), whereas paths with $n > 0$ refer to outer levels.

Of course, in a top-level class expression[2], all paths are expected to denote existing classes, as in the example above.

Paths in DEEPFJIG are *relative*, that is, are computed w.r.t. the current position, whereas, in most mainstream languages supporting nested classes, paths are *absolute*, that is, are computed downward starting from the top-level position, and can be abbreviated by paths starting from a nested position if there is no shadowing. Symmetrically, a realistic language based on DEEPFJIG should allow the programmer to omit **`outer`**s in non ambiguous cases. Formally, a precompilation phase would add **`outer`**s in front of paths of shape $\texttt{outer}^n.C.\pi$, until a scope containing a definition for C is reached. This would lead to a scope resolution analogous to that of Newspeak [BvdAB+10].

Note that the structural type information associated to a class (formally, the type of the class), including, e.g., names and types of members, is directly available for an enclosing basic class, whereas for a nested class, which is defined by a class expression, it is computed combining the class types of its subexpressions, as shown by the following example.

```
C = { int m(){ return 1; } }
  [+] { int k(){ return new outer.C().m(); } }
```

Here, class `C` is defined as the sum of two basic classes. In the latter, the method invocation is well-typed, since **`outer.`**`C` denotes class `C` of the outer level, which has a method `m` provided by the former basic class. The version below, instead,

[2] In DEEPFJIG, differently from Java, FJ and FJIG, a program is a top-level class expression rather than a sequence of class declarations.

```
C = { int m(){ return 1; } }
  [+] { int k(){ return new <>().m(); } }
```

is clearly ill-typed, since the basic class enclosing the invocation does *not* provide a method `m`.

As already pointed out, in DEEPFJIG a class declaration just provides a name which can be used to denote the "semantics" of its right-hand-side class expression. This class expression may contain path occurrences which "refer to the outside", that is, play the role of free variables in the class expression itself. Such path occurrences are called *external*, whereas *internal* path occurrences play the role of bound variables. This difference is reflected when the (semantics of) the class expression is reused, by means of its class name, in a new position, as illustrated by the following example.

```
A = {
  <> mInternal(){ return new <>(); }
  outer.A mExternal(){ return new outer.A(); }
}
B = A
```

In the right-hand-side of the declaration of `A`, which is a basic class, the return type `<>` is internal, since it refers to the basic class itself, whereas the return type **`outer.`**`A` is external, since it refers to the nested class named `A` of the enclosing scope, whose definition accidentally is the same basic class. The declaration of `B` uses the name `A` as a shortcut for (the semantics of) the basic class above, hence is equivalent to the following:

```
A = { /*as before*/ }
B = {
  <> mInternal(){ return new <>(); }
  outer.A mExternal(){ return new outer.A(); }
}
```

Note that `B.mInternal` returns an instance of a new unnamed class, while `B.mExternal` returns an `A`. That is, accordingly with the intuition explained above, when a class is reused in a new position external paths will still denote the "old" class, whereas internal paths will denote a new class.

In the simple example above, since the class is reused in exactly the same scope, this is equivalent to just "copying" code in the new position as it stands. However, in general external path occurrences in the original code need to be modified to preserve the original semantics. For example, consider the class declarations

```
A = {
  B = {
    C = { ... }
    C m1() { ... }
    outer.B.C m2() { ... }
    outer.outer.A.B.C m3() { ... }
  }
}
D = A.B
```

where the three paths which occur as return types denote the same class, which is denoted by `A.B.C` at top level.

At first sight, there is no difference among these three paths, so one could think of normalizing code by always using the shortest path denoting a given class in a given position, e.g., `C` in the example above. However, this makes a difference when code is reused. That is, this code is equivalent to the following:

```
A = { /*as before*/ }
D = {
  C = { ... }
  C m1() { ... }
  outer.A.B.C m2() { ... }
  outer.A.B.C m3() { ... }
}
```

In the definition of D the return type C denotes now a new class, denoted by D.C at top level, whereas the other two return types have been changed in order to still denote the "old" class. This will be expressed by the notation c[from c^s], formally defined in Figure 3, which returns the path obtained by "moving" path c from position c^s (for "source") to the current position.

DEEPFJIG keeps the Java nominal approach, that is, types are class paths (a generalization of class names), and two different paths which denote structurally equivalent classes, or even the same class, are *not* considered equivalent. However, the programmer can explicitly declare a set $c_1 \ldots c_n$ of class paths, introduced by the keyword **implements**, as supertypes of a basic class, as shown below.

```
C = abstract{
   abstract int m1();
   abstract int m2();
}
D = abstract implements outer.C {
  abstract int m1();
  int m2() { return 1 + this.m1(); }
  outer.C m() { return this; }
}
```

In this way we can return **this** as result of method m. The type system checks, for each c_i, that the subtyping relation can be safely assumed, that is, members of c_i are members of the basic class as well (formally, the *class type* of the basic class is a subtype of the class type of c_i). This check is analogous to that on implemented interfaces in Java.

For instance, removing method m1 from D would make the example ill-typed. Note that, differently from Java, where they are implicitly inherited, abstract members must be declared as well, so that it is always possible to compute which are the members provided by a class only from its defining expression.

1.3 Deep composition operators

In DEEPFJIG, composition operators are *deep*, in the sense that they allow to manipulate nested classes at any depth level.

For instance, the sum of two classes "propagates" to their nested classes with the same name, similarly to *deep mixin composition* [OZ05], as shown by the following example:

```
C = {
  N = abstract{
    abstract int n();
    int m(){ return this.n(); }
  }
}
D = C [+] {
    N = {
      int n(){ return 1; }
      abstract int m();
    }
    int k(){ return new N().m(); }
  }
```

Class D is defined as the sum of class C with an unnamed basic class. The effect is that nested class N of C is summed with nested class N of the unnamed basic class. That is, the declaration of D is equivalent to the following:

```
C = { /*as before*/ }
D = {
  N = {
    int n(){ return 1; }
    int m(){ return this.n(); }
  }
  int k(){ return new N().m(); }
}
```

In this way, the resulting class N of D inherits the implementation for methods n and m from N of C and N of the unnamed basic class, respectively.

This example also illustrates the meaning of the modifier **abstract**. As in Java, the effect of the modifier is to forbid the creation of instances of a given class. However, here being abstract is a property of an unnamed class, rather than of a class declaration, accordingly with the DEEPFJIG design principle that a class declaration just gives a name to a class expression. Hence, the modifier is applied to a basic class, and the operators act on the kind (abstract/non abstract) of a class as on other components. Moreover, it is perfectly legal to declare abstract members inside a non abstract class, as shown by nested class N of the unnamed basic class above. The meaning is that the class is *incomplete*, that is, not executable. However, it can be safely used as a library, since it can be completed by composition with another class, as actually happens in the example, where method k of D can correctly create an instance of nested class N of D.

Besides sum, DEEPFJIG provides the following other composition operators, which all modify a single class, taken as first argument: *restrict*, *alias*, *class alias*, *redirect* and *class redirect*. They are illustrated by the following examples.

```
E = {
 C = abstract{
  abstract int n1();
  int n2(){ return 2; }
  int n3(){ return 3; }
 }
 K = {
  int n1(){ return 10; }
  int n2(){ return 20; }
 }
 int m(){ return new K().n1(); }
}

Restrict = E[restrict n2 in C]
Alias = E[alias n2 to n1 in C]
AliasC = E[alias K to C.K]
AliasCSum = Restrict[alias K to C]
Redirect = E[redirect n1 of K to n2]
RedirectC1 = AliasC[redirect K to C.K]
RedirectC2 = AliasC[redirect K to outer.E.K]
```

The restrict operator removes a definition in a nested class, making the corresponding member abstract. Hence we get the following definition:

```
Restrict = {//E[restrict n2 in C]
 C = abstract {
  abstract int n1();
  abstract int n2();//now abstract
  int n3(){ return 3; }
 }
 K = { /*as before*/ }
 int m(){ return new K().n1(); }
}
```

Note that a **restrict** operator for classes makes no sense. On the other hand, a derived operator which recursively makes abstract all fields and methods of a class can be easily defined.

The alias operator duplicates the declaration of an existing field or method of a nested class (including <>), for another field or method of the same class. Hence we get the following definition:

```
Alias = {//E[alias n2 to n1 in C]
 C = abstract {
  int n1(){ return 2; }//now implemented
  int n2(){ return 2; }
  int n3(){ return 3; }
 }
```

```
  K = { /*as before*/ }
  int m(){ return new K().n1(); }
}
```

where the method `C.n1` is now implemented using the implementation of `C.n2`. The method body is duplicated, rather than just invoked, so that, in case the implementation of the original method is changed, the aliased one keeps the original semantics.

The class alias operator adds or modifies a nested class, by duplicating an existing basic class. More precisely, if there is no nested class in the target position, then a new class declaration is inserted, as in the `AliasC` example. Hence we get the following definition:

```
AliasC = {//E[alias K to C.K]
  C = abstract {
    abstract int n1();
    int n2(){ return 2; }
    int n3(){ return 3; }
    K = { int n1() { return 10;}
          int n2() { return 20;} }
  }
  K = { /*as before*/ }
  int m(){ return new K().n1(); }
}
```

If, instead, there is already a nested class in the target position, as in the `AliasCSum` example, then the duplicated class is summed with the existing class. Hence we get the following definition:

```
AliasCSum = {//Restrict[alias K to C]
  C = { int n1(){ return 10; }
        int n2(){ return 20; }
        int n3(){ return 3; } }
  K = { /*as before*/ }
  int m(){ return new K().n1(); }
}
```

where nested class `C` has been obtained by sum, hence has now implementations for `n1` and `n2` copied from `K`.

The redirect operator replaces all the references to a field or method name whose receiver's static type is a given nested class by a different name, and removes its declaration. Hence we get the following definition:

```
Redirect = {//E[redirect n1 of K to n2]
  C = { /*as before*/ }
  K = {
    //int n1(){ return 10; } //removed
    int n2(){ return 20; }
  }
  int m(){ return new K().n2(); }
}
```

where the declaration of `K.n1` has been removed, and the invocation of `K.n1` is now an invocation of `K.n2`.

The class redirect operator replaces all the references to a nested class by a different class, and removes its declaration. Hence we get the following definition:

```
RedirectC1 = {//AliasC[redirect K to C.K]
  C = { /*as before*/ }
  //K = //removed
  int m(){ return new C.K().n1(); }
}
```

where nested class `K` has been removed, and the constructor invocation refers now to class `C.K`.

In all the examples above, paths occurring as arguments of operators are downward paths, that is, they refer to a nested position in the class occurring as first argument of the operator.

However, paths referring to outer classes can occur as source path in the class alias operator, and as target in the class redirect operator. The last example shows the latter case.

```
RedirectC2 = {//AliasC[redirect K to outer.E.K]
  C = { /*as before*/ }
  //K = // removed
  int m(){ return new outer.E.K().n1(); }
}
```

Note that in many cases the application of a composition operator can be impossible or unsafe. For instance, two classes with conflicting definitions for the same member cannot be summed, and the redirect operator can remove a field or method needed to implement a supertype. All these ill-formed applications are prevented by the DEEPFJIG type system, as will be detailed and formalized in Section 3.

Historical excursus The choice of these operators as composition primitives originates from [AZ02], where it was formally shown how to encode all the operators of the Jigsaw framework [Bra92] by sum, freeze and reduct. These three operators were, then, taken as primitives in FJIG [LSZ09a, LSZ09b, LSZ12], where, as in Jigsaw, defined field and methods can be *virtual*, *frozen* or *local*. In DEEPFJIG, instead, defined field and methods are all implicitly virtual, hence we do not include the *freeze* primitive operator which allows to express, e.g., hiding. Moreover, the *reduct* operator, handling maps from names into names, has been replaced by three operators which handle single names (restrict, alias and redirect), which provide the same expressive power and are more convenient for the meta-level which we develop in [Ser11]. The integration of the composition operators with class nesting was never investigated in previous work.

On top of these composition primitives, we can derive many other useful operators. For instance, the **`override`** operator, a variant of sum where conflicts are allowed and the left argument has the precedence, can be defined, by a type-driven translation, as follows:

```
C1[override]C2 ≡
  C1[+](C2[restrict n1 in N1] ... [restrict nk in Nk])
```

where **`restrict`** is applied to all fields or methods with the same name `ni` defined in a nested class (at any depth level) `Ni` in both `C1` and `C2`. Indeed, here override is deep, that is, it propagates to nested classes analogously to sum.

It is possible to define also a **`rename`** operator for nested classes as follows:

```
C[rename COld to CNew] ≡
  C[alias COld to CNew][redirect COld to CNew]
```

If `COld` has nested classes, then we need to recursively apply redirect to these classes.

Renaming of methods and fields can be encoded as follow:

```
C[rename nOld to nNew in N] ≡
  C[alias nOld to nNew in N][redirect nOld of N to nNew]
```

1.4 Expressive power

Expression problem First of all we show the expressive power of DEEPFJIG by considering as "benchmark" the classical *expression problem* (or *extensibility problem*) [Ern04, Tor04, OZ05].

The expression problem can be formulated as follows: we have a datatype defined by a set of *variants*, and we have *processors* which operate on this datatype. The addition of new variants and new processors are the two directions along which the system can be extended. The challenge is to do it in a modular and easy way.

For sake of concreteness, let us consider a `Base` class, modelling arithmetic expressions, defined as follows:

```
Base = {
  Expression = abstract{ abstract String toString(); }
  Num = implements outer.Expression{
    int e; constructor(int e){ this.e = e; }
    String toString(){ return ""+this.e; }
  }
  Sum = implements outer.Expression{
    outer.Expression l, r;
    constructor(outer.Expression l, outer.Expression r){
      this.l = l; this.r = r;
    }
    String toString(){
      return "("+this.l.toString()+
        "+"+this.r.toString()+")"; }
  }
}
```

Note that, here and in other examples, the syntactic convention mentioned at page 5 would allow to omit some **`outer`**s, e.g., to write just `Expression` instead of **`outer`**`.Expression`.

Assume that now Adam wants to add a `UMinus` class. This can be done in this way:

```
AddUMinus = {
  Expression = abstract{ abstract String toString(); }
  UMinus = implements outer.Expression{
    outer.Expression e;
    constructor(outer.Expression e){ this.e = e; }
    String toString(){ return "-"+this.e.toString(); }
  }
}
BaseWithUMinus = Base [+] AddUMinus
```

Bob wants to add an `eval` operator. This can be done in this way:

```
EvalBase = {
  Expression = abstract{ abstract int eval(); }
  Num = abstract implements outer.Expression{
    abstract int e; constructor(int e){}
    int eval(){ return this.e; }
  }
  Sum = abstract implements outer.Expression{
    abstract outer.Expression l, r;
    constructor(outer.Expression l, outer.Expression r){}
    int eval(){ return this.l.eval()+this.r.eval(); }
  }
}
BaseWithEval = Base [+] EvalBase
```

Charles wants to use the work of Adam and Bob to obtain something with both the `UMinus` variant and the `eval` processor. The first step is to define the behaviour of `eval` on the `UMinus` variant.

```
EvalUMinus = {
  Expression = abstract{ abstract int eval(); }
  UMinus = abstract implements outer.Expression{
    abstract outer.Expression e;
    constructor(outer.Expression e){}
    int eval(){ return -this.e.eval(); }
  }
}
```

Now Charles has the following data variants and processors to deal with.

<table>
<tr><th></th><th>constructor</th><th>toString</th><th>eval</th></tr>
<tr><td>Num</td><td colspan="2" rowspan="2">Base</td><td rowspan="2">EvalBase</td></tr>
<tr><td>Sum</td></tr>
<tr><td>UMinus</td><td colspan="2">AddUMinus</td><td>EvalUMinus</td></tr>
</table>

He has two legal ways to compose everything together:

- first `AddUMinus` with `EvalUMinus`, obtaining a fully fledged `UMinus` variant, and then the result with `BaseWithEval`:
  `Solution1 = (AddUMinus [+] EvalUMinus) [+] BaseWithEval;`
- first `EvalBase` with `EvalUMinus`, obtaining a fully fledged `eval` processor, and then the result with `BaseWithUMinus`:
  `Solution2 = (EvalBase [+] EvalUMinus) [+] BaseWithUMinus.`

This solution to the expression problem is very natural and fulfils all the requirements given in [OZ05], that is: extensibility in both dimensions, strong static type safety, no modification or duplication of source code[3], separate compilation, independent extensibility. Among the many solutions existing in the literature, ours is very close to the one in [Ern04], however we use simpler language constructs and type system.

Note that we have no code duplication, in the sense that we *do not have* any duplication of method bodies, which can be seen as "real code". We only need to insert some abstract declarations for required members, which could be alternatively inferred by a type-checker.

What we have done is to "patch" some already existing code. It is possible to do even better if the software is written from the beginning in a fully modular way, that is: for each `Variant` of a `DataType`, we define a class `ConstrVariant`, containing field initializations.

```
ConstrVariant = {
  DataType = abstract{}
  Variant = implements DataType{
   // field declarations
   constructor(...) { ... }
  }
}
```

For each `Variant` and `processor` we define the corresponding processor implementation in a class `ProcessorVariant`.

```
ProcessorVariant = {
  DataType = abstract{ abstract processor(); }
  Variant = abstract implements DataType{
    // abstract field declarations required by processor
    processor(){...}
  }
}
```

Now we have a full grid of processors and variants. For example, for improving modularity, we could have split the `Base` class defined before into four pieces: `ConstrNum`, `ConstrSum`, `ToStringNum`, `ToStringSum`. Analogously, `EvalBase` can be split in `EvalNum` and `EvalSum`, and `AddUMinus` in `ConstrUMinus` and `ToStringUMinus`.

	constructor	toString	eval
Num	ConstrNum	ToStringNum	EvalNum
Sum	ConstrSum	ToStringSum	EvalSum
UMinus	ConstrUMinus	ToStringUMinus	EvalUMinus

This allows the user to take any coherent (that is, where all existing processors are defined over all existing variants) subset of the cells of the grid, and extending the grid is also very

[3]However, code would be expanded by a pre-processor implementing flattening, whereas code expansion could be delayed at invocation time by an implementation generalizing dynamic method look-up, as discussed at the end of Section 1.1.

natural. Analogously, the extension must be coherent, that is, the type system requires to add an entire row or column, to ensure that we are not leaving unmanaged cases.

This possibility of taking only a *subset* of the classes composing a program nicely implements the concept of *scalable-down* architecture [Par78], that is, a software architecture which can be not only easily extended, but also contracted when less functionalities are needed. Note that this means that code size is *truly* reduced, not just that some functionality is hidden as in other approaches.

Generics and `MyType` In DEEPFJIG we can encode generics. Indeed, at the foundational level it has been proved since long time [WV00, AZ02] that module calculi can encode lambda-calculus, and thanks to nesting DEEPFJIG classes play the role of modules with class components, similarly, e.g., to JAVAMOD [AZ01], a module layer for Java classes where an analogous encoding was possible. However, here the encoding is much more natural and does not require additional notions, as shown by the example below.

```
OList = {
 Elem = abstract{ abstract boolean geq(<> other); }
 List = abstract{ abstract <> insert(outer.Elem e); }
 EmptyList = implements outer.List{
  outer.List insert(outer.Elem e){ return new outer.NonEmptyList(e,this); }
 }
 NonEmptyList = implements outer.List{
  outer.Elem e; outer.List tail;
  constructor(outer.Elem e, outer.List tail){ this.e=e; this.tail=tail; }
  outer.List insert(outer.Elem e){
   if(this.e.geq(e))return new <>(e,this);
   return new <>(this.e,this.tail.insert(e));
  }
 }
}
MyElem = {
 int e; constructor(int e){this.e=e;}
 boolean geq(<> other){ return this.e>=other.e; }
}
MyElemOList = OList[redirect Elem to outer.MyElem]
```

The class `OList.List` models an ordered list of `Olist.Elem`, that offers a binary method `geq`. By the redirect operator it is possible to produce an instantiation of `OList` which represents a list of `MyElem`.

The example also shows the binary method `geq` where, as already mentioned, the path `<>` plays the same role of **`MyType`** [BOW98], or **`ThisClass`** of LOOJ [BF04]. **`MyType`** can be used inside a method of a class to refer to the class itself, and, similarly to what happens with **`this`**, is redirected to the proper subclass when the method is inherited. Again, **`MyType`** was already expressible in a previous work on a module layer for Java classes [ALZ06], but here it is smoothly integrated with the overall language design, being just a special case of path.

Note that in both cases there is no true polymorphism, since, as already mentioned, code for each different instantiation is obtained by expanding generic code by flattening. The advantage is that we can keep a standard Java-like type system.

One powerful feature of our approach, that we share with package templates [KMPS09], but not supported by many other proposals, is that we can use the same class more than once in a single class expression. For instance, with the `OList` class we can define a list of lists in the following way:

```
OListOfList=
OList[rename Elem to InnerList]
[+](
 OList
 [+]
 {
 List=abstract{abstract boolean geq(<> other); }
 EmptyList={boolean geq(outer.List){...}}
```

```
 NonEmptyList={boolean geq(outer.List){...}}
 }
 )[rename List to InnerList]
 [rename EmptyList to InnerEmptyList]
 [rename NonEmptyList to InnerNonEmptyList]
```

Refactoring Refactoring tools allow one to perform useful code transformations, notably renaming. When working with a library, it can be useful to keep refactoring operations in the code, so that when the next version of the library is released, it can be seamlessly integrated with the program. For instance, using the rename operator

```
A = { ... C = { ... } }[rename C to D]
B = { ... }
```

allows class `B` to use `A.D` instead of `A.C`.

The code transformation which *moves* a class up or down in the nesting hierarchy can also be encoded as a renaming. For instance:

```
A = { B = { C = { ... } } }[rename B.C to C]
E = { ... }
```

allows class `E` to use `A.C` instead of `A.B.C`.

This approach is different from conventional refactoring, where the tool simply produces the new source. This is a non invasive operation, allowing rollback by simply removing the refactoring code.

AOP Class composition languages and aspect-oriented programming take a different approach: the former construct new classes from existing ones, while the latter modifies the whole program at once. Since in DEEPFJIG "the whole program" is a class expression, we can use composition operators to modify the whole program as well. In this way, the effect is analogous to AOP in many respects: flattening is a code expansion as weaving, anonymous basic classes play the role of advices and composition operators individuate the pointcuts, even though the latter can be specified by a richer language.

Two relevant code modifications allowed by aspects are *execution-around* and *call-around* [KHH+01]: the former replaces execution of a given method, determined by the receiver's dynamic type, the latter replaces invocation of a given method, determined by the receiver's static type.

Consider for instance the following basic class b:

```
{
 A = { int foo(){return 1; } }
 B = implements outer.A{ int foo(){ return 2; } }
 int bar(A a){ return a.foo(); }
 String main(){ return new B().foo()+" "+
    this.bar(new B())+" "+this.bar(new A()); }
}
```

Here a call of `main` produces `"2 2 1"`. Execution-around can be easily emulated by our operators; for example, this AspectJ-like code:

```
int around(): execution(int A.foo()){ return 10; }
```

can be encoded by

```
b[restrict foo in A] [+]{ A = { int foo(){return 10;} }}
```

Now `main` produces `"2 2 10"` since we changed the result of the third call, which is the only one whose receiver has dynamic type `A`. Note that, instead of writing an `A` class with a `foo` method, we could have used an arbitrary named class with an arbitrary named method, and then the **`rename`** operator, to stress that basic classes and composition operators correspond to advices and pointcuts, respectively. However, we find this solution more readable.

Emulating call-around code like

```
int around(): call(int A.foo()) { return 10; }
```

requires a little more effort:

```
(b [+]
  { A = { int foo2(){return 10; } } }[alias A to B]
) [redirect foo of A to foo2]
```

Now `main` produces `"2 10 10"` since we changed the behaviour of all invocations of `foo` whose receiver has static type `A`. The call of `B.foo` is not affected. Note that the **`alias`** is needed to keep `B` subtype of `A`. As in the AOP tradition, this approach does not require to change the source, that is, it is "non invasive". The operation *proceed* can be encoded using the same pattern one can use to emulate *super* calls, that is, as calls to an alias of the original method. *After* and *before* can be encoded by *around* and a call to *proceed*. Other operations, like dynamic pointcuts, are much more complex to express (essentially, the same encoding used by the AspectJ implementation is needed).

2 Formalization

Syntax The syntax of the language is given in Figure 1.

ce	::=		class expression
		b	basic class
		$\mid c$	(class) path
		$\mid ce_1\,[+]\,ce_2$	sum
		$\mid ce\,[\mathbf{restrict}\;i\;\mathbf{in}\;\pi]$	restrict
		$\mid ce\,[\mathbf{alias}\;i^s\;\mathbf{to}\;i^t\;\mathbf{in}\;\pi]$	alias
		$\mid ce\,[\mathbf{alias}\;c^s\;\mathbf{to}\;\pi^t]$	class alias
		$\mid ce\,[\mathbf{redirect}\;i^s\;\mathbf{of}\;\pi\;\mathbf{to}\;i^t]$	redirect
		$\mid ce\,[\mathbf{redirect}\;\pi^s\;\mathbf{to}\;c^t]$	class redirect
b	::=	$ch\{\;k\;\overline{d}\}$	basic class
ch	::=	$\mu\;\mathbf{implements}\;\overline{c}$	class header
k	::=	$kh\{\;\overline{fe}\}$	constructor
kh	::=	$\mathbf{constructor}(\overline{c\;x})$	constructor header
fe	::=	$\mathbf{this}.f = e;$	field expression
d	::=	$fd \mid md \mid cd$	(member) declaration
fd	::=	$\mu\;c\;f;$	field declaration
md	::=	$\mathbf{abstract}\;mh; \mid mh\{\;\mathbf{return}\;e;\}$	method declaration
cd	::=	$C = ce$	class declaration
mh	::=	$c\;m(\overline{c\;x})$	method header
μ	::=	$\epsilon \mid \mathbf{abstract}$	abstract modifier
n	::=	$i \mid C$	(member) name
f, m	::=	i	field name, method name
c	::=	$\overline{\mathbf{outer}}.\pi$	(class) path
π	::=	$\overline{C}$	downward (class) path
e	::=	$x \mid e.[c]f \mid e.[c]m(\overline{e}) \mid \mathbf{new}\;c(\overline{e})$	expression (conventional)
		$\mid c(\overline{fe})$	expression (pre-object)
v	::=	$c(\overline{fv})$	value
fv	::=	$\mathbf{this}.f = v;$	field value
cv	::=	$ch\{\;k\;\overline{fd}\;\overline{md}\;\overline{C = cv}\}$	class value
σ	::=	$\overline{b}$	environment (enclosing classes)

Figure 1 – Syntax

We assume infinite disjoint sets of *class names* C, *instance member names* i (that is, names for field or methods, since nested classes are static members instead), and *variables* x. As in FJ, variables include the special variable **`this`**. We use the bar notation for sequences, e.g., $\overline{d}$ is a sequence of declarations d. We decorate by the "s" (respectively, "t") superscript a metavariable occurrence, e.g., c^s, π^t, to suggest that this occurrence plays the role of a source (respectively, target). The syntax is designed to keep a Java-like flavour as much as possible.

Class expressions are basic classes, (class) paths, or are constructed by composition operators.

A basic class consists in an optional **`abstract`** modifier, a sequence of supertypes, a constructor, and a sequence of (member) declarations.

There is no overloading, hence a class has only one constructor. However, differently from FJ, where this unique constructor has a canonical form, there is no a priori relation among the parameter list and the constructor body, which is a sequence of *field expressions* associating (initialization) expressions to field names.

Field and method declarations are in the style of FJ.

Sequences of supertypes, field expressions, and declarations are considered as sets, that is, order and repetitions are immaterial.

In a well-formed basic class, no instance member name or class name can be declared twice. Hence a sequence of declarations is a map from names into declarations. This implies that, differently from Java, there is no method overloading, and there is no overloading between field and method names. However, for better readability, we will use the metavariable f when a name is used for a field, m for a method. A parameter name cannot be declared twice in a constructor or method header. Finally, a field name cannot appear twice in a sequence of field expressions, hence a sequence $\overline{fe}$ is a map from field names into field expressions. Moreover, there is exactly one field expression in the constructor for each non abstract field.

Since a sequence of declarations $\overline{d}$ is a map, we can use the standard notations $\mathsf{dom}(\overline{d})$, $\overline{d}(n)$, and $\overline{d}\backslash n$, and analogously for other sequences which are maps. .

Expressions in method bodies are similar to those of FJ. We omit cast for simplicity since it is not relevant for our technical treatment. Moreover, field accesses and method invocations are annotated with the static type of the receiver. These annotations can be added during typechecking. However, for simplicity we do not model here two different languages and assume that they are already in source code. This is needed for the redirect operator, see in the following. Finally, expressions include *pre-objects* $c\,(\overline{fe})$, runtime expressions which cannot be written in programmer's code, but are obtained by reducing a constructor invocation. Indeed, since the constructor has no canonical form, we need two different syntactic forms [LSZ09a, LSZ09b, LSZ12], differently from FJ.

Values are objects, that is, pre-objects where all field expressions are (recursively) values.

Class values are basic classes where all nested class definitions are (recursively) class values. Indeed, since operators are deep, they can be applied only to basic classes where all nested class definitions are (recursively) basic classes.

We assume that, in a well-formed top-level class expression, all paths refer to existing classes.

Flattening rules Figure 2 contains the rules defining the flattening relation. The relation is of the form $ce_1 \rightarrow ce_2$, where the unique rule (CTX) reduces the whole program (top-level class expression) by applying a reduction step to either the top-level class expression, or to a class expression appearing as right-hand side of a nested class declaration, at any level of depth. This is formally expressed by the contexts for flattening $\mathcal{CE}^{\mathsf{f}}$, defined in terms of the conventional contexts $\mathcal{CE}$.

$$\mathcal{CE} ::= \Box \mid \mathcal{CE}\,[+]\,ce \mid ce\,[+]\,\mathcal{CE} \mid \mathcal{CE}\,[\,\mathtt{restrict}\ i\ \mathtt{in}\ \pi\,] \mid \mathcal{CE}\,[\,\mathtt{alias}\ i^s\ \mathtt{to}\ i^t\ \mathtt{in}\ \pi\,] \mid \mathcal{CE}\,[\,\mathtt{alias}\ c^s\ \mathtt{to}\ \pi^t\,]$$
$$\mid \mathcal{CE}\,[\,\mathtt{redirect}\ i^s\ \mathtt{of}\ \pi\ \mathtt{to}\ i^t\,] \mid \mathcal{CE}\,[\,\mathtt{redirect}\ \pi^s\ \mathtt{to}\ c^t\,]$$
$$\mathcal{CE}^{\mathrm{f}} ::= \mathcal{CE} \mid \mathcal{CE}[\![\_\{\ \_\_\ C = \mathcal{CE}^{\mathrm{f}}\ \}]\!]$$

$$ce_1 \rightarrow ce_2$$

$$\text{(CTX)}\ \frac{ce_1 \underset{\sigma}{\rightarrow} ce_2}{\mathcal{CE}^{\mathrm{f}}[\![ce_1]\!] \rightarrow \mathcal{CE}^{\mathrm{f}}[\![ce_2]\!]}\ \sigma = \mathsf{env}(ce_1, \mathcal{CE}^{\mathrm{f}})$$

$$ce_1 \underset{\sigma}{\rightarrow} ce_2$$

$$\text{(CLASS-PATH)}\ \frac{}{c \underset{\sigma}{\rightarrow} cv}\ cv = \mathsf{cBody}(\sigma, c)[\mathsf{from}\ c] \qquad \text{(SUM)}\ \frac{}{cv_1\,[+]\,cv_2 \underset{\sigma}{\rightarrow} cv_1 \oplus cv_2}$$

$$\text{(RESTRICT)}\ \frac{}{cv\,[\,\mathtt{restrict}\ i\ \mathtt{in}\ \pi\,] \underset{\sigma}{\rightarrow} cv \ominus_\pi i \oplus_\pi \mathsf{abs}(d)}\ d = \mathsf{dec}(cv, \pi, i)$$

$$\text{(ALIAS)}\ \frac{}{cv\,[\,\mathtt{alias}\ i^s\ \mathtt{to}\ i^t\ \mathtt{in}\ \pi\,] \underset{\sigma}{\rightarrow} cv' \oplus_\pi \mathsf{named}(i^t, d)}\ \begin{array}{l}\mathsf{constr}(cv,\pi) = kh\{\ \overline{fe}\}\\ cv' = \begin{cases} cv \oplus_\pi \mathtt{this}.i^t = e; & \text{if } \overline{fe}(i^s) = e \\ cv & \text{if } i^s \notin \mathsf{dom}(\overline{fe}) \end{cases}\\ d = \mathsf{dec}(cv, \pi, i^s)\end{array}$$

$$\text{(C-ALIAS)}\ \frac{}{cv\,[\,\mathtt{alias}\ c^s\ \mathtt{to}\ \pi^t.C\,] \underset{\sigma}{\rightarrow} cv \oplus_{\pi^t} (C = cv')}\ cv' = \mathsf{cBody}(cv \cdot \sigma, c^s)[\mathsf{from}\ c^s[\mathsf{in}\ \pi^t]]$$

$$\text{(REDIRECT)}\ \frac{}{cv\,[\,\mathtt{redirect}\ i^s\ \mathtt{of}\ \pi\ \mathtt{to}\ i^t\,] \underset{\sigma}{\rightarrow} (cv \ominus_\pi i^s)[i^s \underset{[\pi]}{\rightsquigarrow} i^t]}\ \begin{array}{l} i^s \in \mathsf{names}(cv, \pi)\\ i^s \neq i^t\end{array}$$

$$\text{(C-REDIRECT)}\ \frac{}{cv\,[\,\mathtt{redirect}\ \pi^s.C\ \mathtt{to}\ c^t\,] \underset{\sigma}{\rightarrow} (cv \ominus_{\pi^s} C)[\pi^s.C \rightsquigarrow c^t]}\ \begin{array}{l}\mathsf{noNested}(cv, \pi^s.C)\\ \pi^s.C \neq c^t\end{array}$$

Figure 2 – Flattening rules

The flattening relation for class expressions is of the form $ce_1 \underset{\sigma}{\rightarrow} ce_2$. Indeed, reduction of a class expression takes place in an *environment* $\sigma = b_0 \cdot \ldots \cdot b_n$ which is the stack of all its enclosing basic classes, starting from the directly enclosing, needed to give semantics to external paths. We denote by $\mathsf{env}(ce, \mathcal{CE}^{\mathrm{f}})$ the stack of basic classes enclosing the hole in $\mathcal{CE}^{\mathrm{f}}[\![ce]\!]$, formally:

- $\mathsf{env}(ce, \mathcal{CE}) = \emptyset$
  $\mathsf{env}(ce, \mathcal{CE}[\![ch\{\ k\ \overline{d}\ C = \mathcal{CE}^{\mathrm{f}}\ \}]\!]) = \mathsf{env}(ce, \mathcal{CE}^{\mathrm{f}}) \cdot ch\{\ k\ \overline{d}\ C = \mathcal{CE}^{\mathrm{f}}[\![ce]\!]\}$

Operational versus denotational semantics Before illustrating flattening clauses in detail, let us briefly discuss the style of our formalization. Flattening is an operational (small step) semantics of class expressions which reduces a class expression to a class value (a basic class whose nested classes, at any inner level, are basic classes as well). This corresponds to a "syntactic" interpretation where a class value is interpreted as the class value itself (in the same way as a function declaration can be interpreted as a closure). This syntatic interpretation makes sense in Java-like calculi, where runtime expressions are evaluated in the context of a (syntactic) class table, where the code of the various methods is available. A more "semantic" interpretation of class expressions would be along the lines of the classical denotational model of inheritance introduced in the thesis of William Cook [Coo89], whose extension to Jigsaw operators has been described in Bracha's thesis [Bra92] and formally modeled in detail by

Ancona and Zucca [AZ98]. In this model, roughly, a class is interpreted as a "generator", that is, a function from a (hierarchical in our case) record of functions to a (hierarchical) record of functions. The input record corresponds to abstract and defined methods (considering only methods for simplicity), whereas the output record corresponds to defined methods. The fact that also defined methods are modeled in the input record models the fact that they are virtual. This is the "open" semantics of a class, to be used when classes are combined by composition operators. The "closed" semantics of a class (with no abstract methods) can then be obtained as the least fixed point of the open semantics, and this closed semantics is used as context for evaluating runtime expressions.

Rule (CLASS-PATH) can be applied when the class expression occurring in position c in σ is a class value (DEEPFJIG has a call-by-value semantics). The notation $\mathsf{cBody}(\sigma, c)$ is formally defined by:

- $\mathsf{cBody}(b_0 \cdot \ldots \cdot b_n, \mathtt{outer}^i.\pi) = \mathsf{cBody}(b_i, \pi) \quad 0 \leq i \leq n$
  $\mathsf{cBody}(ch\{\ k\ \overline{d}\ (C = b)\}\ , C.\pi) = \mathsf{cBody}(b, \pi)$
  $\mathsf{cBody}(b, \Lambda) = b$

In this case, c is replaced by cv, a class value obtained from $\mathsf{cBody}(\sigma, c)$, which is in position c w.r.t. the current position, by "moving" all the occurrences of external paths so that they still denote the same class.

The notations $cv[\mathsf{from}\ c^s]$ and $e[\mathsf{from}\ c^s]$, where s stands for "source", meaning "moving class value cv from c^s to the current position", and "moving expression e from c^s to the current position", respectively, are defined in Figure 3. They are defined by an accumulation parameter j, initially set to 0, corresponding to the nesting level.

$cv[\mathsf{from}\ c^s] \quad e[\mathsf{from}\ c^s]$

$cv[\mathsf{from}\ c^s] = cv[\mathsf{from}\ c^s \setminus 1]_0$
$e[\mathsf{from}\ c^s] = e[\mathsf{from}\ c^s]_0$
$ch\{\ k\ \overline{d}\}\ [\mathsf{from}\ c^s]_j = ch[\mathsf{from}\ c^s]_{j+1}\{\ k[\mathsf{from}\ c^s]_{j+1}\ \overline{d}[\mathsf{from}\ c^s]_{j+1}\}$

$$c[\mathsf{from}\ c^s]_j = \begin{cases} \mathtt{outer}^j.(c'[\mathsf{from}\ c^s]) & \text{if } c = \mathtt{outer}^j.c' \\ c & \text{otherwise} \end{cases}$$

$c[\mathsf{from}\ c^s]$

$\mathtt{outer}^n.\pi[\mathsf{from}\ \mathtt{outer}^m.\pi'] = \mathtt{outer}^m.(\pi' \backslash n).\pi$ where

$$C_1.\ \ldots\ .C_k \backslash n = \begin{cases} C_1 \ldots C_{k-n} & \text{if } n \leq k \\ \mathtt{outer}^{n-k} & \text{if } n > k \end{cases}$$

Figure 3 – Auxiliary definitions for moving paths

For simplicity, we omit all trivial propagation clauses, and only report the clauses for a basic class and for a path.

The clause for a basic class simply propagates the operation inside all class components, keeping trace that one nesting level has been added.

The clause for a path is the base case. When a path occurrence is found at some nesting level j, it needs to be moved only if it is external, that is, has form $\mathtt{outer}^j.c'$. In this case, c' is replaced by $c'[\mathsf{from}\ c^s]$, defined in the lower section of the figure. Here, $\pi' \backslash n$ is obtained by removing from π', from right to left, n elements, adding **outer**s if there are not elements enough. For instance, in the example in Section 1.4,

```
A = {
  B = {
    C = { ... }
    C m1() { ... }
    outer.B.C m2() { ... }
    outer.outer.A.B.C m3() { ... }
  }
}
D = A.B
```

paths `outer.B.C` and `outer.outer.A.B.C` are "moved" from position `A.B`, where they are nested at level 0, hence they are external. Applying the definition we obtain `outer.`$(\texttt{B.C}[\mathsf{from}\ \texttt{A}])$ and `outer.`$(\texttt{outer.A.B.C}[\mathsf{from}\ \texttt{A}])$, respectively, hence we get two times `outer.A.B.C`.

The other rules model composition operators.

In rule (SUM), when the arguments of the sum operator are two class values, the operator can be applied, obtaining the sum of the class values, denoted by $cv_1 \oplus cv_2$. This sum is well-defined only if the arguments have the same constructor header and, for each field or method i declared in both arguments, the two declarations have the same kind (field or method) and type, and at most one is non abstract. In this case, the resulting class value has modifier abstract only if both the class values are abstract; the union of the supertypes, the same constructor header, the (necessarily disjoint) union of the field expressions, and the union of the declarations, where two declarations for the same name are merged by keeping the non abstract, if any. Nested classes with the same name are recursively summed. Formally:

- If $cv_i = \mu_i\ \texttt{implements}\ \overline{c}_i\{\ kh\{\ \overline{fe}_i\}\ \overline{d}_i\}$, then
  $cv_1 \oplus cv_2 = \mu\ \texttt{implements}\ \overline{c}_1\ \overline{c}_2\{\ kh\{\ \overline{fe}_1\ \overline{fe}_2\}\ \overline{d}_1 \oplus \overline{d}_2\}$
  where:
  - $\mu = \texttt{abstract}$ iff $\mu_1 = \mu_2 = \texttt{abstract}$
  - $\overline{d}_1 \oplus \overline{d}_2$ is defined by:
    - $\mathsf{dom}(\overline{d}_1 \oplus \overline{d}_2) = \mathsf{dom}(\overline{d}_1) \cup \mathsf{dom}(\overline{d}_2)$
    - $(\overline{d}_1 \oplus \overline{d}_2)(n) = \begin{cases} \overline{d}_1(n) & \text{if } n \in \mathsf{dom}(\overline{d}_1) \setminus \mathsf{dom}(\overline{d}_2) \\ \overline{d}_2(n) & \text{if } n \in \mathsf{dom}(\overline{d}_2) \setminus \mathsf{dom}(\overline{d}_1) \\ \overline{d}_1(n) \oplus \overline{d}_2(n) & \text{if } n \in \mathsf{dom}(\overline{d}_2) \cap \mathsf{dom}(\overline{d}_2) \end{cases}$
    - $\texttt{abstract}\ mh\texttt{;} \oplus \texttt{abstract}\ mh\texttt{;} = \texttt{abstract}\ mh\texttt{;}$
    - $\texttt{abstract}\ mh\texttt{;} \oplus mh\{\ \texttt{return}\ e\texttt{;}\} = mh\{\ \texttt{return}\ e\texttt{;}\} \oplus \texttt{abstract}\ mh\texttt{;} = mh\{\ \texttt{return}\ e\texttt{;}\}$
    - $\texttt{abstract}\ c\ f\texttt{;} \oplus \mu\ c\ f\texttt{;} = \mu\ c\ f\texttt{;} \oplus \texttt{abstract}\ c\ f\texttt{;} = \mu\ c\ f\texttt{;}$
    - $(C = cv_1) \oplus (C = cv_2) = C = (cv_1 \oplus cv_2)$

In rule (RESTRICT), the operator replaces the definition of member i by the corresponding abstract declaration. We denote by $cv \ominus_\pi n$ the class value obtained from cv by removing member n of class π (if n is a field, then its initialization expression is removed as well), by $cv \oplus_\pi d$ the class value obtained from cv by adding declaration d in class π, by $\mathsf{abs}(d)$ the abstract version of the declaration d, by $\mathsf{dec}(cv, \pi, n)$ the declaration for name n in nested class π in cv, by $\mathsf{named}(n, d)$ the declaration equal to d except that the declared name is n. Formally:

- $ch\{\ k\ \overline{d}\ (C = cv)\} \ominus_{C.\pi} n = ch\{\ k\ \overline{d}\ (C = cv \ominus_\pi n)\}$
  $ch\{\ kh\{\ \overline{fe}\}\ \overline{d}\} \ominus_\Lambda n = ch\{\ kh\{\ \overline{fe} \setminus n\}\ \overline{d} \setminus n\}$
- $ch\{\ k\ \overline{d}\ (C = cv)\} \oplus_{C.\pi} d = ch\{\ k\ \overline{d}\ (C = cv \oplus_\pi d)\}$
  $ch\{\ k\ \overline{d}\} \oplus_\Lambda d = ch\{\ k\ \overline{d} \oplus d\}$

- $\mathsf{abs}(\mu\ c\ f\texttt{;}) = \texttt{abstract}\ c\ f\texttt{;}$
  $\mathsf{abs}(\texttt{abstract}\ mh\texttt{;}) = \mathsf{abs}(mh\{\ \texttt{return}\ e\texttt{;}\}) = \texttt{abstract}\ mh\texttt{;}$

- $\mathsf{dec}(ch\{\ k\ \overline{d}\ (C = cv)\}, C.\pi, n) = \mathsf{dec}(cv, \pi, n)$
  $\mathsf{dec}(ch\{\ k\ \overline{d}\ d\}, \Lambda, n) = d$ with $d = \mathsf{named}(n, \_)$

- $\mathsf{named}(i, \mu\ c\ f\texttt{;}) = \mu\ c\ i\texttt{;}$
  $\mathsf{named}(i, \texttt{abstract}\ c\ m(\overline{c\ x})\texttt{;}) = \texttt{abstract}\ c\ i(\overline{c\ x})\texttt{;}$
  $\mathsf{named}(i, c\ m(\overline{c\ x})\{\ \texttt{return}\ e\texttt{;}\}) = c\ i(\overline{c\ x})\{\ \texttt{return}\ e\texttt{;}\}$
  $\mathsf{named}(C, C' = ce) = C = ce$

In rule (ALIAS), the operator adds a definition for field or method i^t, for "target", in class π, by duplicating that existing for i^s, for "source", in the same class. If i^s is a field, then the initialization expression is duplicated as well. We denote by $\mathsf{constr}(cv, \pi)$ the constructor of class π in cv, and by $cv \oplus_\pi fe$ the class value obtained from cv by adding field expression fe in the constructor of π. Formally:

- $\mathsf{constr}(cv, \pi) = k$ if $\mathsf{cBody}(cv, \pi) = \_\{\ k\ \_\}$

- $ch\{\ k\ \overline{d}\ (C = cv)\} \oplus_{C.\pi} fe = ch\{\ k\ \overline{d}\ (C = cv \oplus_\pi fe)\}$
  $ch\{\ kh\{\ \overline{fe}\}\ \overline{d}\} \oplus_\Lambda fe = ch\{\ kh\{\ \overline{fe}\ fe\}\ \overline{d}\}$

In rule (C-ALIAS), the operator adds a definition for nested class C in class π^t, by duplicating that existing for c^s, which must be a class value. If in position π^t there is already a nested class C, then the duplicated class value is summed with the existing class. Analogously to rule (CLASS-PATH), the duplicated class value needs to be modified by "moving" all the occurrences of external class paths from the source position c^s to the target position. However, here the target position is a descendant π^t of the current position (Λ), rather than the current position itself. Hence, the movement must take place from $c^s[\mathsf{in}\ \pi^t]$, the (shortest) path which denotes *in* π^t the class denoted by c^s in the current position, or "c^s as seen *in* position π^t". Formally:

- $c^s[\mathsf{in}\ \Lambda] = c^s$
  $C.\pi^s[\mathsf{in}\ C.\pi^t] = \pi^s[\mathsf{in}\ \pi^t]$
  $c^s[\mathsf{in}\ C.\pi^t] = (\texttt{outer}.c^s)[\mathsf{in}\ \pi^t]$ if $c^s \neq C.\_$.

To see a path in a descendant $C.\pi^t$ of the current position, if the path is a descendant of child node C as well, that is, of form $C.\pi^s$, then we only have to see π^s in π^t (second clause). Otherwise, this corresponds to take child node C as current position, and to see $\texttt{outer}.c^s$ in π^t (third clause).

Note that $c[\mathsf{in}\ \pi^t] = c'$ implies $c'[\mathsf{from}\ \pi^t] = c$; anyway, there can be many c'' such that $c''[\mathsf{from}\ \pi^t] = c$. For example $\texttt{C.D.A}[\mathsf{in}\ \texttt{C.D}] = \texttt{A}$ and $\texttt{A}[\mathsf{from}\ \texttt{C.D}] = \texttt{C.D.A}$ but also $\texttt{outer.D.A}[\mathsf{from}\ \texttt{C.D}] = \texttt{C.D.A}$.

Note also that the source can be an outer class, that is, code to be duplicated can be taken from the outside, whereas of course the target, being code to be modified, cannot. The converse situation takes place for the class redirect operator, see below.

In rule (REDIRECT), the operator replaces references to existing field or method i^s of class π by references to i^t. The declaration of i^s is removed, so i^s and i^t have to be different. We denote by $\mathsf{names}(cv, \pi)$ the set of names declared in nested class π of cv, and by $cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t]$ the class value obtained from cv by replacing i^s with i^t in field accesses/method invocations whose receiver has static type π. Formally:

- $\mathsf{names}(cv, \pi) = \mathsf{dom}(\overline{d})$ if $\mathsf{cBody}(cv, \pi) = \_\{\ \_\ \overline{d}\}$
- $cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t] = cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_\Lambda$
  $(C = cv)[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\pi'} = C = (cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\pi'.C})$
  $.[c]i[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\pi'} = \begin{cases} .[c]i^t & \text{if } c[\mathsf{from}\ \pi'] = \pi \text{ and } i = i^s \\ .[c]i & \text{otherwise} \end{cases}$

In rule (C-REDIRECT), the operator replaces references to existing nested class C of class π^s by references to c^t. The declaration of C is removed, so $\pi^s.C$ and c^t have to be different. Redirect is only defined from a class that contains no nested classes (that is, $\mathsf{noNested}(cv, \pi^s.C)$ holds), hence can be safely removed. It is trivial to define a derived operator which redirects a class with nested classes by performing a sequence of redirect applications. Here $cv[\pi^s \rightsquigarrow c^t]$ is the class value obtained from cv by replacing π^s with c^t in type annotations and **`new`** expressions. Formally:

$cv[\pi^s \rightsquigarrow c^t] = cv[\pi^s \rightsquigarrow c^t]_\Lambda$
$(C = cv)[\pi^s \rightsquigarrow c^t]_\pi = C = (ce[\pi^s \rightsquigarrow c^t]_{\pi.C})$
$c[\pi^s \rightsquigarrow c^t]_\pi = \begin{cases} c^t[\mathsf{in}\ \pi] & \text{if } c[\mathsf{from}\ \pi] = \pi^s \\ c & \text{otherwise} \end{cases}$

The notations $cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t]$ and $cv[\pi^s \rightsquigarrow c^t]$ are defined by an accumulation parameter π corresponding to the nested class where the occurrence (of field or method name and path, respectively) is found. For simplicity, we omit all trivial propagation clauses, and only report the clauses for a basic class and for the base case.

The clause for a basic class simply propagates the operation inside all class components, keeping trace that one more nested class has been entered.

The base case for $cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t]$ takes place when a field access/method invocation (here generically indicated by $.[c]i$) is encountered where i is the member name i^s to be and the annotation c (receiver's static type) denotes π from the current position π'. In this case i^s is replaced by i^t.

The base case for $cv[\pi^s \rightsquigarrow c^t]$ takes place when a path occurrence π is encountered which denotes π^s from the current position π. In this case π^s is replaced by c^t as seen in the current position π.

Dependency relation In order to prevent flattening to get stuck, we must forbid, informally, cyclic reuse of code. In inheritance-based languages, this corresponds to require the inheritance relation to be acyclic. In our framework, this requirement is formalized as follows. We denote by $\mathsf{nested}(ce)$ the set of pairs consisting of an environment and a class name which correspond to a nested class in ce, formally:

- $\sigma, C \in \mathsf{nested}(ce)$ iff
  $ce = \mathcal{CE}^{\mathrm{f}}[\![b]\!]$ and $\sigma = b \cdot \mathsf{env}(b, \mathcal{CE}^{\mathrm{f}})$ with $b = \_\{\ \_\_\ C = \_\}$

In a well-formed class expression ce, the relation $\xrightarrow{\mathsf{dep}}$, defined in Figure 4, over $\mathsf{nested}(ce)$, is required to be acyclic. Informally, $\sigma_1, C_1 \xrightarrow{\mathsf{dep}} \sigma_2, C_2$ holds if, in order to reduce nested class σ_1, C_1, we need (a nested class of) nested class σ_2, C_2, which has not been reduced to a class value yet. Hence, a cyclic dependency would make reduction stuck. We use the following notation:

- $\sigma[\mathsf{in}\ c]$ is the environment σ "as seen in position c". Formally:
  $(b \cdot \sigma)[\mathsf{in}\ \mathtt{outer}.c] = \sigma[\mathsf{in}\ c]$
  $(b \cdot \sigma)[\mathsf{in}\ C.\pi] = (b(c) \cdot b \cdot \sigma)[\mathsf{in}\ \pi]$
  $\sigma[\mathsf{in}\ \Lambda] = \sigma$

$\sigma_1, C_1 \xrightarrow{\mathsf{dep}} \sigma_2, C_2$

$$\text{(DIRECT-DEP)}\ \frac{}{\sigma, C \xrightarrow{\mathsf{dep}} \sigma[\mathsf{in}\ c], C_1}\ \begin{array}{l}\sigma = \_\{\ \_\_\ C = \mathcal{CE}[\![c.C_1 \dots C_k]\!]\}\ \cdot\ \_ \text{ or}\\ \quad \sigma = \_\{\ \_\_\ C = \mathcal{CE}[\![\_[\,\mathtt{alias\ outer}.c.C_1 \dots C_k\ \mathtt{to}\ \_\,]]\!]\}\ \cdot\ \_\\ \mathsf{cBody}(\sigma, c.C_1) \neq cv\\ \mathsf{cBody}(\sigma, c.C_1.C_2) \text{ not defined or } k = 1\end{array}$$

$$\text{(PROP-DEP)}\ \frac{\sigma_1[\mathsf{in}\ C_1], \_ \xrightarrow{\mathsf{dep}} \sigma_2, C_2}{\sigma_1, C_1 \xrightarrow{\mathsf{dep}} \sigma_2, C_2} \qquad \text{(TRANS-DEP)}\ \frac{\sigma_1, C_1 \xrightarrow{\mathsf{dep}} \sigma_2, C_2 \quad \sigma_2, C_2 \xrightarrow{\mathsf{dep}} \sigma_3, C_3}{\sigma_1, C_1 \xrightarrow{\mathsf{dep}} \sigma_3, C_3}$$

Figure 4 – Dependency relation

Reduction Reduction models execution of a *main expression* e in an environment σ. Formally, the reduction arrow has form $e_1 \underset{\sigma}{\rightarrow} e_2$. This models the fact that a *main* method could belong to an arbitrarily nested class.

$e_1 \underset{\sigma}{\rightarrow} e_2$

$$\text{(FIELD-ACCESS)}\ \frac{}{c\,(\overline{fv})\,.f \underset{\sigma}{\rightarrow} v}\ \overline{fv}(f) = v$$

$$\text{(INVK)}\ \frac{}{c\,(\overline{fv})\,.m\,(\overline{v}) \underset{\sigma}{\rightarrow} e[\mathsf{from}\ c][\overline{v}/\overline{x}][^{c\,(\overline{fv})}/_{\mathtt{this}}]}\ \mathsf{mBody}(\sigma, c, m) = \langle \overline{x}, e\rangle$$

$$\text{(OBJ-CREATION)}\ \frac{}{\mathtt{new}\ c\,(\overline{v}) \underset{\sigma}{\rightarrow} c\,(\overline{fe}[\mathsf{from}\ c][\overline{v}/\overline{x}])}\ \begin{array}{l}\mathsf{kBody}(\sigma, c) = \langle \overline{x}, \overline{fe}\rangle\\ \mathsf{nonAbs}(\sigma, c)\end{array}$$

Figure 5 – Reduction rules

Reduction rules are straightforward, and formally defined in Figure 5. The only significant difference w.r.t. FJ is that in rule (INVK) expression e is found in position c, so class paths inside e must be moved to denote the same class value in the current position, and analogously in rule (OBJ-CREATION). We use straightforward functions mBody, kBody and nonAbs which, for a given class, return parameters and body of methods and of the constructor, and check whether the class is abstract. Formally:
if $\mathsf{cBody}(\sigma, c) = \mu\ \mathtt{implements}\ \_\{\ k\ \overline{d}\}$

- $\mathsf{mBody}(\sigma, c, m) = \langle x_1 \dots x_n, e\rangle$ if $\overline{d}(m) = c\ m(c_1\ x_1, \dots, c_n\ x_n)\{\ \mathtt{return}\ e\mathtt{;}\}$
- $\mathsf{kBody}(\sigma, c) = \langle x_1 \dots x_n, \overline{fe}\rangle$ if $k = \mathtt{constructor}(\,c_1\ x_1, \dots, c_n\ x_n)\{\ \overline{fe}\}$
- $\mathsf{nonAbs}(\sigma, c)$ holds iff $\mu \neq \mathtt{abstract}$.

3 Type system

Types and type environments are defined in Figure 6.

Analogously to what happens for flattening, all typing judgements have on the left a class type environment, which is a stack of class types $ct_0 \cdot \ldots \cdot ct_n$, where ct_0 is the type of the directly enclosing class, ct_1 the type of the `outer` class, and so on. A class type is a tuple consisting of the kind (abstract or non abstract), the supertypes, the constructor type, and a

Δ	::=	$\overline{ct}$	class type environment
ct	::=	$[\mu \mid \overline{c} \mid kt \mid \overline{dt}]$	class type
kt	::=	$\overline{c}$	constructor type
dt	::=	$i{:}\mu\ it \mid C{:}ct$	declaration type
it	::=	$c \mid \overline{c}{\to}c$	field or method type
Γ	::=	$\overline{x{:}c}$	parameter type environment

Figure 6 – Types and type environments

map from field/method names to their kinds (abstract or non abstract) and types, and from class names to class types.

Typing rules for environments, basic classes and well-formedness of class types are given in Figure 7. They are straightforward. The typing judgment for environments is used in Theorem 11 and Theorem 18.

Note that in rule (BASIC-T) the constructor body, the method bodies and the nested classes are typechecked in the class type environment obtained by pushing the type of the basic class on the stack of class types. The judgement $\Delta \vdash ct$, see rule (WF-CLASS-TYPE), means that ct is well-formed w.r.t. Δ, that is, the subtyping relation induced by the supertypes in (all nested class types) in ct can be safely assumed. Note that we have to move $\mathsf{cType}(ct \cdot \Delta, c_i)$ from c_i to properly check structural subtyping, defined by rule (STRUCTURAL-S) in Figure 9.

We use the following notations:

- $\mathsf{exists}(\Delta, c_1 \ldots c_n)$ holds if , for all $i \in 1..n$, c_i denotes an existing class in Δ, formally $\mathsf{cType}(\Delta, c_i)$ is defined
- $\mathsf{defFields}(\Delta, c)$ are the declaration types corresponding to non abstract fields of class c in Δ, formally:
  $\mathsf{defFields}(\Delta, c) = \overline{f{:}c}$
  if $\mathsf{cType}(\Delta, c) = [\_ \mid \_ \mid \_ \mid \overline{f{:}c}\ \overline{f{:}\mathbf{abstract}\ c}\ \overline{m{:}\mu\ \overline{c}{\to}c}\ \overline{C{:}ct}]$
- $\mathsf{cType}(\Delta, c)$ and $ct[\mathsf{from}\ c]$ are analogous to $\mathsf{cBody}(\sigma, c)$ and $cv[\mathsf{from}\ c]$, respectively, but work over class types.

Typing rules for composition operators are given in Figure 8. They are similar to corresponding flattening rules.

In some cases, to check that an operator can be safely applied it is necessary to check for well-formedness of the resulting type, since the application of the operator can break the subtyping relation, notably: sum, alias and class alias can add a field or method to a nested class declared as supertype, redirect can remove a field or method needed to implement a supertype, and class redirect can replace a nested class declared as supertype with another one with more fields or methods.

Besides the checks performed in rule (C-REDIRECT), in (C-REDIRECT-T) additional checks need to be performed to ensure that the class path c^t can safely replace the class denoted by $\pi^s.C$. This is formally expressed by the subtyping relation $\Delta \vdash c^t \lhd ct^s$, defined in Figure 9.

We use the following notations:

- $ct_1 \oplus ct_2$, $ct \ominus_\pi n$ and $ct \oplus_\pi dt$ are analogous to $cv_1 \oplus cv_2$, $cv \ominus_\pi n$ and $cv \oplus_\pi d$, respectively, but work over class types

$\vdash \sigma : \Delta$

$$\text{(ENV-T)}\ \frac{ct_{i+1} \cdot \ldots \cdot ct_n \vdash b_i : ct_i \quad \forall i \in 0..n}{\vdash \sigma : \Delta} \quad \begin{array}{l}\sigma = b_0 \cdot \ldots \cdot b_n \\ \Delta = ct_0 \cdot \ldots \cdot ct_n\end{array}$$

$\Delta \vdash ce : ct$

$$\text{(BASIC-T)}\ \frac{ct \cdot \Delta \vdash k : kt \quad ct \cdot \Delta \vdash \overline{d} : \overline{dt} \quad ct \cdot \Delta \vdash ct}{\Delta \vdash \mu\ \mathbf{implements}\ \overline{c}\{\ k\ \overline{d}\}\ : ct} \quad \begin{array}{l}\mathsf{exists}(ct \cdot \Delta, \overline{c}) \\ ct = [\mu \mid \overline{c} \mid kt \mid \overline{dt}]\end{array}$$

$\Delta \vdash k : kt$

$$\text{(CONS-T)}\ \frac{\begin{array}{c}\Delta; x_1{:}c'_1, \ldots, x_n{:}c'_n \vdash e_i : c''_i \quad \forall i \in 1..k \\ \Delta \vdash c''_i \leq c_i \quad \forall i \in 1..k\end{array}}{\Delta \vdash kh\{\ \mathbf{this}.f_1 = e_1\mathbf{;}\ \ldots\ \mathbf{this}.f_k = e_k\mathbf{;}\} : c'_1 \ldots c'_n} \quad \begin{array}{l}\mathsf{exists}(\Delta, c'_i) \quad \forall i \in 1..n \\ kh = \mathbf{constructor}(\ c'_1\ x_1, \ldots, c'_n\ x_n) \\ \mathsf{defFields}(\Delta, \Lambda) = f_1{:}c_1, \ldots, f_k{:}c_k\end{array}$$

$\Delta \vdash d : dt$

$$\text{(FIELD-T)}\ \frac{}{\Delta \vdash (\mu\ c\ f\mathbf{;}) : (f{:}\mu\ c)} \quad \mathsf{exists}(\Delta, c)$$

$$\text{(ABS-METHOD-T)}\ \frac{}{\Delta \vdash (\mathbf{abstract}\ mh\mathbf{;})\ :\ (m{:}\mathbf{abstract}\ c_1 \ldots c_n \to c_0)} \quad \begin{array}{l}mh = c_0\ m(c_1\ x_1, \ldots, c_n\ x_n) \\ \mathsf{exists}(\Delta, c_i) \quad \forall i \in 0..n\end{array}$$

$$\text{(METHOD-T)}\ \frac{\Delta; \mathbf{this}{:}\Lambda, x_1{:}c_1, \ldots, x_n{:}c_n \vdash e : c \quad \Delta \vdash c \leq c_0}{\Delta \vdash mh\{\ \mathbf{return}\ e\mathbf{;}\}\ :\ (m{:}c_1 \ldots c_n \to c_0)} \quad \begin{array}{l}mh = c_0\ m(c_1\ x_1, \ldots, c_n\ x_n) \\ \mathsf{exists}(\Delta, c_i) \quad \forall i \in 0..n\end{array}$$

$$\text{(CLASS-T)}\ \frac{\Delta \vdash ce : ct}{\Delta \vdash (C = ce) : (C{:}ct)}$$

$\Delta \vdash ct$

$$\text{(WF-CLASS-TYPE)}\ \frac{\begin{array}{c}ct \cdot \Delta \vdash ct_j \quad \forall j \in 1..k \\ ct \leq (\mathsf{cType}(ct \cdot \Delta, c_i)[\mathsf{from}\ c_i]) \quad \forall i \in 1..n\end{array}}{\Delta \vdash ct} \quad \begin{array}{l}ct = [\_ \mid c_1 \ldots c_n \mid \_ \mid \overline{dt}] \\ \overline{dt} = \overline{fd}\ \overline{md}\ C_1{:}ct_1 \ldots C_k{:}ct_k\end{array}$$

Figure 7 – Typing rules for environments, basic classes and well-formed class types

$\Delta \vdash ce : ct$

$$\text{(CLASS-PATH-T)}\ \frac{}{\Delta \vdash c : ct}\ ct = \mathsf{cType}(\Delta, c)[\mathsf{from}\ c]$$

$$\text{(SUM-T)}\ \frac{\Delta \vdash ce_1 : ct_1 \quad \Delta \vdash ce_2 : ct_2 \quad \Delta \vdash ct_1 \oplus ct_2}{\Delta \vdash ce_1\ [+]\ ce_2 : ct_1 \oplus ct_2}$$

$$\text{(RESTRICT-T)}\ \frac{\Delta \vdash ce : ct}{\Delta \vdash ce\ [\mathbf{restrict}\ i\ \mathbf{in}\ \pi\ ] : ct \ominus_\pi i \oplus_\pi \mathsf{abs}(dt)}\ dt = \mathsf{decType}(ct, \pi, i)$$

$$\text{(ALIAS-T)}\ \frac{\Delta \vdash ce : ct \quad \Delta \vdash ct \oplus_\pi \mathsf{named}(i^t, dt)}{\Delta \vdash ce\ [\mathbf{alias}\ i^s\ \mathbf{to}\ i^t\ \mathbf{in}\ \pi\ ] : ct \oplus_\pi \mathsf{named}(i^t, dt)}\ dt = \mathsf{decType}(ct, \pi, i^s)$$

$$\text{(C-ALIAS-T)}\ \frac{\Delta \vdash ce : ct \quad \Delta \vdash ct \oplus_{\pi^t} (C{:}\mu\ ct')}{\Delta \vdash ce\ [\mathbf{alias}\ c^s\ \mathbf{to}\ \pi^t.C\ ] : ct \oplus_{\pi^t} (C{:}\mu\ ct')}\ ct' = \mathsf{cType}(ct \cdot \Delta, c^s)[\mathsf{from}\ c^s[\mathsf{in}\ \pi^t]]$$

$$\text{(REDIRECT-T)}\ \frac{\Delta \vdash ce : ct \quad \Delta \vdash ct \ominus_\pi i^s}{\Delta \vdash ce\ [\mathbf{redirect}\ i^s\ \mathbf{of}\ \pi\ \mathbf{to}\ i^t\ ] : ct \ominus_\pi i^s}\ \begin{array}{l}\mathsf{mType}(ct, \pi, i^s) = \mathsf{mType}(ct, \pi, i^t)\\ i^s \neq i^t\end{array}$$

$$\text{(C-REDIRECT-T)}\ \frac{\Delta \vdash ce : ct \quad \Delta \vdash (ct \ominus_{\pi^s} C)[\pi^s.C \rightsquigarrow c^t] \quad (ct \cdot \Delta)[\mathsf{in}\ \pi^s] \vdash c^t[\mathsf{in}\ \pi^s] \lhd \mathsf{cType}(ct, \pi^s.C)[\pi^s.C \rightsquigarrow c^t]}{\Delta \vdash ce\ [\mathbf{redirect}\ \pi^s.C\ \mathbf{to}\ c^t\ ] : (ct \ominus_{\pi^s} C)[\pi^s.C \rightsquigarrow c^t]}\ \begin{array}{l}\pi^s.C \neq c^t\\ \mathsf{noNested}(ct, \pi^s.C)\end{array}$$

Figure 8 – Typing rules for composition operators

- $\mathsf{decType}(ct, \pi, n)$, $\mathsf{abs}(dt)$ and $\mathsf{named}(n, dt)$ are analogous to $\mathsf{dec}(cv, \pi, n)$, $\mathsf{abs}(d)$ and $\mathsf{named}(n, d)$, respectively, but work over declaration types
- $\mathsf{mType}(\Delta, c, i)$ is the type of field or method i in class c in Δ. Formally:
  $\mathsf{mType}(\Delta, c, i) = it$ if $\mathsf{cType}(\Delta, c) = [\_ \mid \_ \mid \_ \mid \overline{dt}\ i{:}\mu\ it]$
- $ct[\pi^s \rightsquigarrow c^t]$ and $\mathsf{noNested}(ct, \pi)$ are analogous to $cv[\pi^s \rightsquigarrow c^t]$ and $\mathsf{noNested}(cv, \pi)$, respectively, but work over class types
- $\Delta[\mathsf{in}\ c]$ is analogous to $\sigma[\mathsf{in}\ c]$, but works over class type environments.

Rules for subtyping relations are given in Figure 9.

The subtyping relation $\Delta \vdash c^t \lhd ct^s$, defined by rule (REDIRECT-T), holds if a path denoting a class whose type is ct^s can be safely redirected to c^t, that is, c^t can be used by clients all the ways the source class is used. That is: c^t is a nominal subtype of the declared supertypes, the class type denoted by c^t is a structural subtype of ct^s, and if the source class is non abstract, then the target is non abstract as well, and the constructor types are the same.

$\Delta \vdash c^t \lhd ct^s$

$$\text{(REDIRECT-S)}\ \frac{\begin{array}{c}\Delta \vdash c^t \leq c_i \quad \forall i \in 1..n \\ [\mu^t \mid \overline{c}^t \mid kt^t \mid \overline{dt}^t] \leq [\mu^s \mid \overline{c}^s \mid kt^s \mid \overline{dt}^s]\end{array}}{\Delta \vdash c^t \lhd [\mu^s \mid \overline{c}^s \mid kt^s \mid \overline{dt}^s]}\ \begin{array}{l}\mathsf{cType}(\Delta, c^t)[\mathsf{from}\ c^t] = [\mu^t \mid \overline{c}^t \mid kt^t \mid \overline{dt}^t] \\ \overline{c}^t = \mathtt{outer}.c_1 \ldots \mathtt{outer}.c_n \\ \mu^s = \epsilon \text{ implies } \mu^t = \epsilon \text{ and } kt^t = kt^s\end{array}$$

$\Delta \vdash c_1 \leq c_2$

$$\text{(DIRECT-S)}\ \frac{}{\Delta \vdash c \leq (c_i[\mathsf{from}\ c])}\ \mathsf{impl}(\Delta, c) = c_1 \ldots c_n$$

$$\text{(REFL-S)}\ \frac{}{\_ \vdash c \leq c} \qquad \text{(TRANS-S)}\ \frac{\Delta \vdash c_1 \leq c_2 \quad \Delta \vdash c_2 \leq c_3}{\Delta \vdash c_1 \leq c_3}$$

$ct_1 \leq ct_2$

$$\text{(STRUCTURAL-S)}\ \frac{}{ct_1 \leq ct_2}\ \begin{array}{l}ct_1 = [\_ \mid \_ \mid \_ \mid (i_1{:}\_\ it_1) \ldots (i_n{:}\_\ it_n)\overline{C{:}ct}] \\ ct_2 = [\_ \mid \_ \mid \_ \mid (i_1{:}\_\ it_1) \ldots (i_n{:}\_\ it_n)\overline{dt}]\end{array}$$

Figure 9 – Subtyping rules

The other subtyping relations are conventional nominal subtyping among paths and width structural subtyping among class types, respectively. We use the notation $\mathsf{impl}(\Delta, c)$, that denotes the declared supertypes of class c in type environment Δ. Formally:
$\mathsf{impl}(\Delta, c) = \overline{c}$ if $\mathsf{cType}(\Delta, c) = [\_ \mid \overline{c} \mid \_ \mid \_]$,

Straightforward typing rules for expressions are given in Figure 10. We use notations $\mathsf{nonAbs}(\Delta, c)$ and $\mathsf{kType}(\Delta, c)$, that are analogous to $\mathsf{nonAbs}(\sigma, c)$ and $\mathsf{constr}(\sigma, c)$, respectively.

Note that in order to use a type c' as declared in position c, we need to use $c'[\mathsf{from}\ c]$.

The type system is sound w.r.t. flattening, that is, a well-typed top-level class expression ce always reduces in some steps to a well-typed class value.

$\boxed{\Delta;\Gamma \vdash e : c}$

$$\text{(VAR-T)}\ \frac{}{\_;\Gamma \vdash x : c}\ \Gamma(x) = c \qquad \text{(FIELD-ACCESS-T)}\ \frac{\Delta;\Gamma \vdash e : c}{\Delta;\Gamma \vdash e.f : c'[\mathsf{from}\ c]}\ \mathsf{mType}(\Delta, c, f) = c'$$

$$\text{(INVK-T)}\ \frac{\begin{array}{c}\Delta;\Gamma \vdash e : c \\ \Delta;\Gamma \vdash \overline{e} : \overline{c} \\ \Delta \vdash \overline{c} \leq (\overline{c}'[\mathsf{from}\ c])\end{array}}{\Delta;\Gamma \vdash e.m(\overline{e}) : c'[\mathsf{from}\ c]}\ \mathsf{mType}(\Delta, c, m) = \overline{c}' \to c'$$

$$\text{(NEW-T)}\ \frac{\begin{array}{c}\Delta;\Gamma \vdash \overline{e} : \overline{c} \\ \Delta \vdash \overline{c} \leq (\overline{c}'[\mathsf{from}\ c])\end{array}}{\Delta;\Gamma \vdash \mathbf{new}\ c(\overline{e}) : c}\ \begin{array}{l}\mathsf{nonAbs}(\Delta, c) \\ \mathsf{kType}(\Delta, c) = \overline{c}'\end{array}$$

$$\text{(OBJ-T)}\ \frac{\begin{array}{c}\Delta;\Gamma \vdash e_i : c_i \quad \forall i \in 1..n \\ \Delta \vdash c_i \leq (c_i'[\mathsf{from}\ c]) \quad \forall i \in 1..n\end{array}}{\Delta;\Gamma \vdash c(\overline{fe}) : c}\ \begin{array}{l}\mathsf{nonAbs}(\Delta, c) \\ \overline{fe} = \mathbf{this}.f_1 = e_1; \ldots \mathbf{this}.f_n = e_n; \\ \mathsf{defFields}(\Delta, c) = f_1{:}c_1', \ldots, f_n{:}c_n'\end{array}$$

Figure 10 – Typing rules for expressions

Theorem 1 (Soundness w.r.t. flattening)**.** *If* $\Lambda \vdash ce : ct$*, then* $ce \xrightarrow{*} cv$*, and* $\Lambda \vdash cv : ct$*.*

In order to express soundness of the type system w.r.t. (expression) reduction, we need to introduce two notations:

- σ^v is an *environment value*, that is, a stack of class values
- $\mathsf{isComplete}(\Delta)$ holds if the class type environment Δ is *complete*, that is, non abstract class types (at any depth level) do not contain abstract fields or methods. Formally:

  $\mathsf{isComplete}(ct_1 \ldots ct_n)$ iff for all $i \in [1..n]\mathsf{isComplete}(ct_i)$
  $\mathsf{isComplete}([\mu \mid \_ \mid \_ \mid \overline{dt}])$ iff $\overline{dt}(i) = i{:}\mathbf{abstract}\ \_$ implies $\mu = \mathbf{abstract}$, and $\overline{dt}(C) = C{:}ct$ implies $\mathsf{isComplete}(ct)$.

The type system is sound w.r.t. (expression) reduction, that is, given a well-typed environment σ with a complete class type environment, the reduction of a closed expression well-typed w.r.t. σ never goes stuck. Class values with an incomplete class type can be safely used as a library, but are not executable.

Theorem 2 (Soundness)**.** *If* $\vdash \sigma : \Delta$*,* $\mathsf{isComplete}(\Delta)$*,* $\Delta;\emptyset \vdash e_1 : c$ *and* $e_1 \xrightarrow[\sigma]{*} e_2$ *then either* e_2 *is a value or* $e_2 \xrightarrow[\sigma]{} \_$*.*

Proofs of the results are in the Appendix.

4 Discussion and related work

In the following, we will stress the distinguishing features of our approach w.r.t other proposals in the literature.

True language of class expressions Perhaps the most important difference between our work and most other mechanisms for building new classes from existing ones is that DEEPFJIG, as FJIG, provides a true language of class expressions. Note once again the difference between our declaration:

```
B = { /*some class expression*/}
A = B
```

and the analogous in Java and most existing proposals:

```
class A extends B {}
```

The meaning of our declaration is just

```
B = { /*some class expression*/}
A = { /*some class expression*/}
```

whereas in the Java-like declaration the use of name `B` is relevant, not only since inheritance implies nominal subtyping, but also, e.g., in case `B` has static members.

That is, the semantics and the type system of our language fully achieve the *substitutivity* principle: a class expression can be replaced by an equivalent one in any context without affecting the overall semantics or well-typedness.

For this reason, in DEEPFJIG two different class paths which denote the same class are incompatible types. In this way the meaning of a class expression is fully context-independent. For instance, in the following example, types of `f3` and `f4` are incompatible, even though both denote, in this context, class `A.A`. Indeed, they could denote different classes if the definition of `A.A.A` would occur in an arbitrary context.

```
{
  A = {
    A = {
      A = {
        A = {} ...
        A f1; <> f2; outer f3; outer.outer.A f4;
      }
    }
  }
}
```

This example also shows that, thanks to "outer" types, we can refer to all enclosing classes. In other proposals, like Jx [NCM04] discussed below, the type `This` corresponds to our empty path `<>`, and `A` corresponds to our `A`, but there is no type corresponding to **`outer`** and **`outer.outer.`**`A`.

No virtual superclasses Many languages provide some mechanism to declare classes inside other classes, that is, support nested classes. In such languages, a (nested) class name can be either *directly used*, that is, as type annotation or instance generator, or used for building new classes, that is, for *code reuse*. In the following example (written with Java syntax and scoping rules)

```
class A{
  class B{ int mb(){return 1;} }
  class C extends B{}
  int ma1(){ return new B().mb();}
  int ma2(){ return new C().mb();}
}
class AA extends A{
  class B{ int mb(){return 2;} }
}
```

`new` `B()` and **`new`** `C()` are examples of the former kind of use, whereas **`extends`** `B` is an example of the latter. Both kinds of use can be either virtual or not. Notably, the invocation **`new`** `AA().ma1()` will evaluate to `2` if the former kind of use is virtual, otherwise to `1`, and

the invocation **new** `AA().ma2()` will evaluate to `2` if both kinds of use are virtual, otherwise to `1`.

Different choices make sense for a language design, in detail:

- In order to see nested classes as virtual members, in the sense of the Jigsaw framework, that is, exactly as methods are virtual members, both kinds of use should be virtual.
- In Java nested classes have static binding, as fields and static methods. That is, nested classes are not virtual.
- In the first versions of BETA [MMPN93] only the former kind of use is virtual.
- In the literature on family polymorphism (also known as virtual classes) [Ern01, EOC06, ISV05, IV07, ISV08], deep mixin composition [OZ05, Hut06] usually also the latter kind of use is virtual, and this feature is called "supporting virtual superclasses". Virtual superclasses can also be emulated by C++ templates, as shown in the work on mixin layers [SB01].
- In DEEPFJIG, only the former kind of use is virtual, that is, we do not support virtual superclasses, since flattening removes all the information about the way a class was defined, that is, (in a class expression) references to other classes (formally, subexpressions which are paths) are all implicitly frozen.

This approach allows us to keep a simple type system, while keeping, as our examples show, the main advantages. Indeed, supporting virtual superclasses would require the type of `A` to maintain information about the fact that `C` extends `B`. This exposes the inheritance hierarchy and, as Bracha pointed out [Bra92], breaks modularity. This problem is already present in all the proposals supporting virtual superclasses (that usually offer only the **extends** operator) and would be even worse in our (richer) composition language. A compositional type system should likely use constraints and type variables as in [MW05].

On the other hand, providing virtual semantics for the former kind of use fits very well flattening and substitutivity principle. For example, in

```
A = { B = {...} B m(){ return new B(); } }
AA = A [+] { B = {...} }
```

class `AA` is a subclass of class `A`, and expression **new** `B()` in `AA` clearly refers to the resulting class `AA.B`. Note that this semantics is sound since our composition operators never remove members from a class, except for the redirect operator, which, however, also replaces references to the removed member.

Subclassing is different from subtyping Another design choice concerns the interaction between virtual classes and inheritance. In proposals where subtyping and subclassing coincide, a naive approach is unsound. Consider for instance the following code, written in .FJ syntax [ISV05, ISV08].

```
class A{
  static class B{int f1;}
  int k(.B x){ return x.f1;}
}
class AA extends A{
  static class B{int f2;}
  int k(.B x){return x.f2+new .B().f2; }
}
```

The notation `.B` stands for a relative path, that is, `B` as visible in the current scope. Whereas in Java declaring in a subclass a nested class with the same name of a nested of the superclass

has only the effect of hiding parent's declaration, in [ISV05, ISV08], as in other approaches [NQM06, EOC06, BOW98, IV07], class `AA.B` *further extends* [IV07] `A.B`, that is, is implicitly considered a subclass of `A.B`, adding the field `f2`. This is analogous to the behavior of our deep sum operator. Consider now the following code:

```
new AA().k(new AA.B())//well-typed
new A().k(new A.B())//well-typed
A a=new AA(); //well-typed assuming AA subtype of A
a.k(new A.B())//runtime error: A.B.f2 does not exist
```

To ensure soundness, in .FJ, indeed, the last method invocation is considered ill-typed, even though `AA.B` is a subtype of `A.B`. This example can be rewritten, with minor syntactical changes, in the other approaches [NQM06, EOC06, BOW98, IV07]. That is, many authors recognize the need to break the coincidence of subtyping and subclassing in some controlled way. In these works, this means that subtyping and subclassing coincide whenever this does not directly lead to unsoundness, and this is ensured by additional checks on method invocations. This choice allows to be closer to the Java **`extends`** relation.

As in package templates [KMPS09], we choose a more radical approach, that is, subclassing and subtyping are totally unrelated and the latter should be explicitly declared by the programmer (as explained at page 7), hence in the example there is no a priori subtyping relation between `A` and `AA`.

Many approaches offering implicit subtyping relation impose that the **`extends`** relation can only be declared between nested classes of the same outer class (i.e., *family*), while here we have no such limitation.

Nested classes are class members Another criterion that can be used to classify proposals on nested classes is whether they are members of instances [Ern01, OZ05, Hut06, CDNW07, BvdAB+10] or members of classes [ISV05, IV07, ISV08, NCM04, NQM06]. The former choice is more expressive, but requires a complex type system usually involving dependent types. Our model follows the latter choice, mainly resembling nested classes of C++ and C#, and static nested classes of Java.

Jigsaw-like operators We are not aware of any previous language or calculus supporting both Jigsaw-like operators and nesting. Anyway, some works go in a similar direction, for example Nystrom et al. develop Jx [NCM04], a language providing an expressive power similar to our deep **`override`** operator, defined at page 10. They also introduce an "hypothetical extension of Jx with abstract types" allowing an encoding of generics very similar to ours with the **`redirect`** operator. However, in our work **`redirect`** can be applied to any type, not only to ad-hoc "abstract types", not to be confused with abstract classes.

J& [NQM06] is an extension of Jx with a composition mechanism similar to our sum operator; however, it still misses the expressiveness required to encode the examples of generics, AOP and refactoring.

More advanced mechanisms are provided by *package templates* [KMPS09], which are collections of classes. However, differently from standard Java packages where an import clause only provides a shorter name for a class which is in any case part of the current class table, here importing a package template has the effect of adding its classes, either as they stand or with some modification, to the class table, by a precompilation step analogous to our flattening. That is, package templates are a generative way to build new classes. It is also possible to import a package template into another one. This provides an easy and modular way to construct complex package templates. When two package templates are imported at the same time, classes with the same name are merged in a way similar to our hierarchical sum. Moreover, many other operators similar to ours can be used: it is possible to rename

classes and methods, to manually select an implementation for a method in case of conflict, as we do by the restrict operator, and generic types can be fixed, similarly to what we have shown in the example about emulating generics. In the original proposal generic types offer fixed generic constraints, while our encoding is more flexible, since it transparently uses a nested class. In recent work [AK12] they extend the original proposal in order to provide generic constraints as first class entities of the template, in the same way that classes and interfaces are. Finally, as in our approach, there is a complete separation between code reuse and subtyping. In summary, using a very different technique, package templates provides an expressive power very similar to the one of DEEPFJIG. However, they do not support nesting of classes, but only provides a sort of first level nesting.

As already mentioned, sum operator in DEEPFJIG is similar to deep mixin composition [Ern99b, OZ05, Hut06], also supported by the Scala language, and family polymorphism [EOC06, ISV05, IV07, ISV08]. However, our sum is symmetric, with a more flexible explicit conflict resolution, whereas implicit precedence rules for method invocation become hard to maintain in the case of mixin chains, and even more complex for deep mixin composition. This is effectively shown in [Sch05], where Schärli performs a refactoring of the Smalltalk library using traits, that offer only symmetric composition.

No dependent types Both Jx [NCM04] and the virtual classes of Ernst [EOC06] make uses of dependent types. As in ^FJ [IV07], instead, we do not need sophisticated types. In the following example (a simplified version of an example from [NCM04], rephrased in our syntax):

```
A = { B = {...} int m(B b){...} }
A2 = A [override] {
  B = { ... int y;}
  int m(B b){ ... b.y ... }
}
```

`A2` is not a subtype of `A`, and *cannot* be declared to be a subtype of `A` (that is, it would be a type error to write `A2 = implements A ...`), since the parameter types of method `m` are non compatible (they denote the classes `A.B` and `A2.B`, respectively). In other words, we have *no* notion of "family". However, code which works uniformly over "families", for instance a method invocation `x.m(y)` which works with `x` and `y` of (static) type `A`, `A.B` and `A2`, `A2.B`, can be obtained as shown below:

```
C = {
  X = abstract{ abstract int m(Y y);}
  Y = abstract{}
  int k(){ return new X().m(new Y()); }
}
CA=C[redirect Y to outer.A.B][redirect X to outer.A]
CA2=C[redirect Y to outer.A2.B][redirect X to outer.A2]
```

In ^FJ [IV07], functionalities which work uniformly over families can be obtained using generics.

Compositional type analysis Finally, we stress that our approach to type checking is very different from, e.g., C++ templates, where type checking is deferred until after template instantiation, at which moment the whole code resulting from the instantiation is analyzed by the standard C++ type checker. On the contrary, we are able to entirely type check a DEEPFJIG program (top-level expression) *before* flattening. More precisely, as base step we type check (by a standard Java-like type checker) all the basic classes occurring in the program (as formally expressed by rules in Figure 7). By these checks, we detect all Java standard errors, such as, e.g., invoking a non existing method or passing an argument of the wrong type. On top of this, each application of a composition operator, e.g., a sum, is analyzed

with additional checks. By these checks, we only detect errors due to the application of the operator (not to intrinsic ill-formedness of the arguments), for instance, in the case of a sum, that there are two conflicting definitions for the same member. In summary, our type analysis is *compositional*.

5 Conclusion

We have defined the language DEEPFJIG, which smoothly integrates operators for modular composition of classes with nesting, achieving a great expressive power by simple ingredients: essentially, Java-like (nested) classes where inheritance has been replaced by a powerful set of operators (sum, restrict, alias and redirect) inspired by Bracha's Jigsaw framework and trait-based languages. In this way, a single class becomes an adequate unit of reuse, since, embodying a whole hierarchy of classes which can be manipulated by the operators, it plays also the role of a module (or a component if you wish).

There are many possible directions for further research.

Since the first submission of this paper, an extension we have already developed [Ser11] is that with a meta-level, analogously to what has been done for FJIG in [SZ10]. The expressive power of the meta-level, together with the capability of representing a whole component as a single class, allows one to encapsulate a library within a single meta-expression. Moreover, the possibility offered by the meta-level to write classes whose structure depends on an external source, like a database table, having nested classes is generalized to a whole hierarchy, as one can extract from a whole database or XML schema.

We have already mentioned at page 4 an open research direction towards a realistic language, that is, the design of a mechanism for object initialization which can be smoothy integrated with symmetric composition, notably in presence of side effects. As mentioned there, object creation expressions, as in Javascript, Emerald [RTL+91] or Grace [BBN10]), could be a possible solution.

Another important, partly related, issue are visibility levels. The language FJIG [LSZ09a, LSZ09b, LSZ12] also includes *frozen* and *private* members, and, correspondingly, operators such as *freeze* and *hide*, as in the original Jigsaw framework [Bra92]. However, the interaction of hiding with nesting is not trivial. Consider, for instance, the following two classes:

```
C = {
  A = { int ma(){return 1;}}
  B = { int mb(){return new outer.A().ma();}}
  }
D = C [hide ma in A]
```

We could expect this code to reduce by flattening to:

```
C = //as before
D = {
  A = { private int ma(){return 1;}}
  B = { int mb(){return new outer.A().ma();}}
  }
```

But this would be unsound, since `D.B` would call the private method `D.A.ma`. A possible solution could be a "many level" private modifier, where a **`private`**`[`n`]` member is visible only in the first n enclosing classes.

Other interesting issues concern adding static members and annotations.

Acknowledgments Partially supported by MIUR DISCO - Distribution, Interaction, Specification, Composition for Object Systems. We warmly thank the anonymous referees of FOOL, PPPJ and JOT for many useful comments on previous versions of this work.

References

[AK12] Eyvind W. Axelsen and Stein Krogdahl. Adaptable generic programming with required type specifications and package templates. In Robert Hirschfeld, Éric Tanter, Kevin J. Sullivan, and Richard P. Gabriel, editors, *AOSD'12 - Aspect-oriented software development*, pages 83–94. ACM Press, 2012. doi:10.1145/2162049.2162060.

[ALZ06] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Flexible type-safe linking of components for Java-like languages. In *JMLC'06 - Joint Modular Languages Conference*, volume 4228 of *Lecture Notes in Computer Science*, pages 136–154. Springer, 2006. doi:10.1007/11860990_10.

[AZ98] Davide Ancona and Elena Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, August 1998. doi:10.1017/S0960129598002576.

[AZ01] Davide Ancona and Elena Zucca. True modules for Java-like languages. In J.L. Knudsen, editor, *ECOOP'01 - European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 354–380. Springer, 2001. doi:10.1007/3-540-45337-7_19.

[AZ02] Davide Ancona and Elena Zucca. A calculus of module systems. *Journ. of Functional Programming*, 12(2):91–132, 2002. doi:10.1017/S0956796801004257.

[BBN10] Andrew Black, Kim B. Bruce, and James Noble. Panel: designing the next educational programming language. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *SPLASH/OOPSLA Companion - Object-Oriented Programming, Systems, Languages, and Applications*, pages 201–204. ACM Press, 2010. doi:10.1145/1869542.1869574.

[BDG07] Viviana Bono, Ferruccio Damiani, and Elena Giachino. Separating type, behavior, and state to achieve very fine-grained reuse. In *9th Intl. Workshop on Formal Techniques for Java-like Programs*, 2007. Available from: http://www.di.unito.it/~damiani/papers/ftfjp07.html.

[BDG08] Viviana Bono, Ferruccio Damiani, and Elena Giachino. On traits and types in a Java-like setting. In *TCS'08 - IFIP Int. Conf. on Theoretical Computer Science*. Springer, 2008. doi:10.1007/978-0-387-09680-3_25.

[BDNW08] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Computer Languages, Systems and Structures*, 34(2-3):83–108, 2008. doi:10.1016/j.cl.2007.05.003.

[BF04] Kim B. Bruce and J. Nathan Foster. LOOJ: Weaving LOOM into Java. In *ECOOP'04 - Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 389–413, 2004. doi:10.1007/978-3-540-24851-4_18.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutmire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1998*, pages 183–200. ACM Press, October 1998. doi:10.1145/286936.286957.

[BOW98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *ECOOP'98 - European Conference on Object-Oriented*

Programming, volume 1445 of *Lecture Notes in Computer Science*, pages 523–549, 1998. doi:10.1007/BFb0054106.

[Bra92] Gilad Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992. Available from: http://www.bracha.org/jigsaw.ps.

[BvdAB+10] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in Newspeak. In Theo D'Hondt, editor, *ECOOP'10 - Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 405–428. Springer, 2010. doi:10.1007/978-3-642-14107-2_20.

[CDNW07] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: a simple virtual class calculus. In Brian M. Barry and Oege de Moor, editors, *AOSD'07 - Aspect-oriented software development*, volume 208, pages 121–134. ACM Press, 2007. doi:10.1145/1218563.1218578.

[Coo89] William R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Dept. Comp. Sci., Brown University, 1989. Available from: http://www.cs.utexas.edu/~wcook/papers/thesis/cook89.pdf.

[CSZ10] Andrea Corradi, Marco Servetto, and Elena Zucca. DeepFJig - Modular composition of nested classes. In *FOOL 2010 - Intl. Workshop on Foundations of Object-Oriented Languages*, 2010. Available from: http://ecee.colorado.edu/~siek/FOOL2010/corradi.pdf.

[CSZ11] Andrea Corradi, Marco Servetto, and Elena Zucca. DeepFJig - Modular composition of nested classes. In Christian Wimmer and Christian W. Probst, editors, *PPPJ'11 - Principles and Practice of Programming in Java*, ACM International Proceedings Series, pages 101–110. ACM Press, 2011. doi:10.1145/2093157.2093172.

[EOC06] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *ACM Symp. on Principles of Programming Languages 2006*, volume 41, pages 270–282. ACM Press, 2006. doi:10.1145/1111037.1111062.

[Ern99a] Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Århus, Denmark, 1999. Available from: http://cs.au.dk/~eernst/papers/thesis.ps.

[Ern99b] Erik Ernst. Propagating class and method combination. In *ECOOP'99 - European Conference on Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 67–91. Springer, 1999. doi:10.1007/3-540-48743-3_4.

[Ern01] Erik Ernst. Family polymorphism. In J.L. Knudsen, editor, *ECOOP'01 - European Conference on Object-Oriented Programming*, number 2072 in Lecture Notes in Computer Science, pages 303–326. Springer, 2001. doi:10.1007/3-540-45337-7_17.

[Ern04] Erik Ernst. The expression problem, scandinavian style. In *MASPEGHI 2004 - Mechanisms for specialization, generalization and inheritance*, 2004. Available from: http://www.i3s.unice.fr/maspeghi2004/final-version/e_ernst.pdf.

[GJSB05] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java language specification*. The Java series. Addison-Wesley, third edition, 2005.

[Hut06] DeLesley Hutchins. Eliminating distinctions of class: using prototypes to model virtual classes. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2006)*, pages 1–20. ACM Press, 2006. doi:10.1145/1167473.1167475.

[IPW99] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, pages 132–146. ACM Press, 1999. doi:10.1145/320384.320395.

[ISV05] Atsushi Igarashi, Chieri Saito, and Mirko Viroli. Lightweight family polymorphism. In Kwangkeun Yi, editor, *APLAS 2005 - Asian Symposium on Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 161–177. Springer, 2005. doi:10.1017/S0956796807006405.

[ISV08] Atsushi Igarashi, Chieri Saito, and Mirko Viroli. Lightweight family polymorphism. *Journ. of Functional Programming*, 18(3):285–331, 2008. doi:10.1017/S0956796807006405.

[IV07] Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2007)*, pages 113–132. ACM Press, 2007. doi:10.1145/1297027.1297037.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *ECOOP'01 - European Conference on Object-Oriented Programming*, number 2072 in Lecture Notes in Computer Science, pages 327–353. Springer, 2001. doi:10.1007/3-540-45337-7_18.

[KMPS09] Stein Krogdahl, Birger Møller-Pedersen, and Fredrik Sørensen. Exploring the use of package templates for flexible re-use of collections of related classes. *Journ. of Object Technology*, 8(7):59–85, 2009. doi:10.5381/jot.2009.8.7.a1.

[LSZ09a] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Featherweight Jigsaw - a minimal core calculus for modular composition of classes. In Sophia Drossopoulou, editor, *ECOOP'09 - Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*. Springer, 2009. doi:10.1007/978-3-642-03013-0_12.

[LSZ09b] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Flattening versus direct semantics for Featherweight Jigsaw. In *FOOL'09 - Intl. Workshop on Foundations of Object-Oriented Languages*, 2009. Available from: http://www.cs.cmu.edu/~aldrich/FOOL09/lagorio.pdf.

[LSZ12] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Featherweight Jigsaw - replacing inheritance by composition in Java-like languages. *Information and Computation*, 214:86–111, 2012. doi:10.1016/j.ic.2012.02.004.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

[MW05] Henning Makholm and J. B. Wells. Type inference, principal typings, and let-polymorphism for first-class mixin modules. In Olivier Danvy and Benjamin C. Pierce, editors, *Intl. Conf. on Functional Programming 2005*, pages 156–167. ACM Press, 2005. doi:10.1145/1086365.1086386.

[NCM04] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2004)*, pages 99–115. ACM Press, 2004. doi:10.1145/1028976.1028986.

[NQM06] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: nested intersection for scalable software composition. *SIGPLAN Not.*, 41(10):21–36, October 2006. doi:10.1145/1167515.1167476.

[OZ05] Martin Odersky and Matthias Zenger. Independently extensible solutions to the expression problem. In *FOOL'05 - Intl. Workshop on Foundations of Object-Oriented Languages*, 2005. Available from: http://homepages.inf.ed.ac.uk/wadler/fool/program/final/10/10_Paper.pdf.

[Par78] David L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press. doi:10.1109/TSE.1979.234169.

[RTL+91] Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software - Practice and Experience*, 21(1):91–118, 1991. doi:10.1002/spe.4380210107.

[SB01] Yannis Smaragdakis and Don S. Batory. Mixin-based programming in C++. In Gregory Butler and Stan Jarzabek, editors, *GCSE'00 - Generative and Component-Based Software Engineering*, volume 2177 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2001. doi:10.1007/3-540-44815-2_12.

[Sch05] Nathanael Schärli. *Traits — Composing Classes from Behavioral Building Blocks*. PhD thesis, University of Bern, February 2005. Available from: http://www.iam.unibe.ch/~scg/Archive/PhD/schaerli-phd.pdf.

[SDNB03] 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. doi:10.1007/978-3-540-45070-2_12.

[Ser11] Marco Servetto. *MetaFJig - A Meta-Circular Composition Language for Java-like Classes*. PhD thesis, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 2011. Available from: http://bart.disi.unige.it/bibliography/files/servettoMarco.pdf.

[SZ10] Marco Servetto and Elena Zucca. MetaFJig - A meta-circular composition language for Java-like classes. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *ACM SIGPLAN Conference on Object-Oriented*

Programming, Systems, Languages and Applications (OOPSLA 2010), pages 464–483. ACM Press, 2010. doi:10.1145/1869459.1869498.

[Tor04] Mads Torgersen. The expression problem revisited. In Martin Odersky, editor, *ECOOP'04 - Object-Oriented Programming*, number 3086 in Lecture Notes in Computer Science, pages 123–143. Springer, 2004. doi:10.1007/978-3-540-24851-4_6.

[WV00] J. B. Wells and René. Vestergaard. 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. doi:10.1007/3-540-46425-5_27.

About the authors

Andrea Corradi Contact him at andreac@unstable.it.

Marco Servetto Contact him at marco.servetto@ecs.vuw.ac.nz, or visit http://ecs.victoria.ac.nz/Main/MarcoServetto.

Elena Zucca Contact her at zucca@disi.unige.it, or visit http://www.disi.unige.it/person/ZuccaE/.

A Results

Soundness w.r.t. flattening follows from termination, progress, and subject reduction theorems below. Note that progress relies on the assumption that the dependency relation $\xrightarrow{\text{dep}}$ is acyclic.

In order to prove termination of flattening, we formally define the dimension of a class expression:

$\dim(\_\{\ \_\overline{fd}\ \overline{md}\ \overline{cd}\}) = \dim(\overline{cd})$
$\dim(cd_1 \ldots cd_n) = \dim(cd_1) + \ldots + \dim(cd_n)$
$\dim(\_ = ce) = \dim(ce)$
$\dim(ce_1\,[+]\,ce_2) = \dim(ce_1) + \dim(ce_2) + 1$
$\dim(ce\,[\texttt{restrict\_in\_}]) = \dim(ce) + 1$
$\ldots$
$\dim(c) = 1$

Lemma 3. *If* $ce_1 \xrightarrow[\sigma]{} ce_2$ *then* $\dim(ce_1) > \dim(ce_2)$.

Proof. By cases on the flattening rules. We show only one case:

Case $\text{(SUM)}\dfrac{}{cv_1\,[+]\,cv_2 \xrightarrow[\sigma]{} cv_1 \oplus cv_2}$

$\dim(cv_1\,[+]\,cv_2) = 1$ and $\dim(cv_1 \oplus cv_2) = 0$.

The other cases are analogous. □

Theorem 4 (Termination of flattening)**.** *If* $ce_1 \to ce_2$ *then* $\dim(ce_1) > \dim(ce_2)$.

Proof. Only rule $\text{(CTX)}\dfrac{ce_1 \xrightarrow[\sigma]{} ce_2}{\mathcal{CE}^{\mathrm{f}}[\![ce_1]\!] \to \mathcal{CE}^{\mathrm{f}}[\![ce_2]\!]}\ \sigma = \mathsf{env}(ce_1, \mathcal{CE}^{\mathrm{f}})$

reduces a top-level class expression. By Lemma 3, we have that $\dim(ce_1) > \dim(ce_2)$. We can conclude by definition of dim. □

The following lemmas are needed to prove progress w.r.t. flattening.

Lemma 5. *If* $\vdash \mathcal{CE}^{\mathrm{f}}[\![ce]\!] : ct$ *and* $\sigma = \mathsf{env}(ce, \mathcal{CE}^{\mathrm{f}})$*, then* $\vdash \sigma : \Delta$ *and* $\Delta \vdash ce : \_$.

In the next lemma we use the following definition of redex.

$$\begin{aligned} r ::= \quad & c \mid cv_1\,[+]\,cv_2 \mid cv\,[\texttt{restrict}\ i\ \texttt{in}\ \pi\,] \mid cv\,[\texttt{alias}\ i^s\ \texttt{to}\ i^t\ \texttt{in}\ \pi\,] \\ & \mid cv\,[\texttt{alias}\ c^s\ \texttt{to}\ \pi^t\,] \mid cv\,[\texttt{redirect}\ i^s\ \texttt{of}\ \pi\ \texttt{to}\ i^t\,] \mid cv\,[\texttt{redirect}\ \pi^s\ \texttt{to}\ c^t\,] \end{aligned}$$

Lemma 6 (Progress w.r.t. $ce_1 \xrightarrow[\sigma]{} ce_2$)**.** *If* $\vdash \sigma : \Delta$ *and* $\Delta \vdash r : ct$ *then either* $r \xrightarrow[\sigma]{} \_$ *or* $\sigma = \_\{\ \_\_\ C = \mathcal{CE}[\![r]\!]\}\ \cdot\ \_$ *and* $\sigma, C \xrightarrow{\text{dep}} \_, \_$.

Proof. By cases. We show only two cases:

Case $r = cv_1\,[+]\,cv_2$

typed by $\text{(SUM-T)}\dfrac{\Delta \vdash cv_1 : ct_1 \quad \Delta \vdash cv_2 : ct_2 \quad \Delta \vdash ct_1 \oplus ct_2}{\Delta \vdash cv_1\,[+]\,cv_2 : ct_1 \oplus ct_2}$

We get the thesis by application of rule $\text{(SUM)}\dfrac{}{cv_1\,[+]\,cv_2 \xrightarrow[\sigma]{} cv_1 \oplus cv_2}$, since $cv_1 \oplus cv_2$ is well-defined. Indeed, since $\Delta \vdash cv_1 : ct_1$ and $\Delta \vdash cv_2 : ct_2$ are typed using (BASIC-T) and $ct_1 \oplus ct_2$ is well-defined, we know that the constructor of cv_1 is equal to the constructor of cv_2, and by definition of $\overline{dt}_1 \oplus \overline{dt}_2$ we know that also $\overline{d}_1 \oplus \overline{d}_2$ is well-defined, where $\overline{d}_i$ are the declarations of cv_i and $\overline{dt}_i$ are the declaration types of ct_i.

Case $r = c$

typed by $\text{(CLASS-PATH-T)}\dfrac{}{\Delta \vdash c : ct}\ ct = \mathsf{cType}(\Delta, c)[\mathsf{from}\ c]$

In this case either $cv = \mathsf{cBody}(\sigma, c)[\mathsf{from}\ c]$ holds, and we get the thesis by application of $\text{(CLASS-PATH)}\dfrac{}{c \underset{\sigma}{\rightarrow} cv}\ cv = \mathsf{cBody}(\sigma, c)[\mathsf{from}\ c]$, or not, and we get the thesis since $\sigma \xrightarrow{\mathsf{dep}} \_$.

□

Theorem 7 (Progress w.r.t. flattening)**.** *If* $\emptyset \vdash ce : ct$ *then either* ce *is a class value or* $ce \rightarrow \_$.

Proof. If ce is not a class value, then $ce = \mathcal{CE}^{\mathrm{f}}[\![r]\!]$. Set $\sigma = \mathsf{env}(\mathcal{CE}^{\mathrm{f}}, r)$. By Lemma 5 we know that r is well-typed. If σ is empty, then the thesis hold by Lemma 6. Otherwise by construction $\sigma = \_\{\ \_\_\ C = \mathcal{CE}[\![r]\!]\} \cdot \_$. Since $\xrightarrow{\mathsf{dep}}$ is acyclic, we can choose $\mathcal{CE}^{\mathrm{f}}$ in such a way that $\sigma, C \xrightarrow{\mathsf{dep}} \_, \_$ does not hold.Hence, the thesis holds by Lemma 6. □

The following lemmas are needed to prove subject-reduction w.r.t. flattening.

Lemma 8 ($\mathcal{CE}^{\mathrm{f}}$ substitution)**.** *If* $\Delta \vdash ce_1 : ct$*,* $\Delta \vdash ce_2 : ct$*, and* $\Delta \vdash \mathcal{CE}^{\mathrm{f}}[\![ce_1]\!] : ct'$*, then* $\Delta \vdash \mathcal{CE}^{\mathrm{f}}[\![ce_2]\!] : ct'$.

Proof. By straightforward structural induction on $\mathcal{CE}^{\mathrm{f}}$. □

Lemma 9 (Subject reduction w.r.t. $ce_1 \underset{\sigma}{\rightarrow} ce_2$)**.** *If* $\vdash \sigma : \Delta$*,* $\Delta \vdash ce_1 : ct$ *and* $ce_1 \underset{\sigma}{\rightarrow} ce_2$ *then* $\Delta \vdash ce_2 : ct$.

Proof. By cases on the flattening rules. We show only one case:

Case $\text{(SUM)}\dfrac{}{cv_1\,[+]\,cv_2 \underset{\sigma}{\rightarrow} cv_1 \oplus cv_2}$

typed by $\text{(SUM-T)}\dfrac{\Delta \vdash cv_1 : ct_1 \quad \Delta \vdash cv_2 : ct_2 \quad \Delta \vdash ct_1 \oplus ct_2}{\Delta \vdash cv_1\,[+]\,cv_2 : ct_1 \oplus ct_2}$

We get the thesis since $\Delta \vdash cv_1 : ct_1$ and $\Delta \vdash cv_2 : ct_2$ are typed using (BASIC-T), and $ct_1 \oplus ct_2$ is analogous to $cv_1 \oplus cv_2$. □

Theorem 10 (Subject reduction w.r.t. flattening)**.** *If* $\vdash ce_1 : ct$ *and* $ce_1 \rightarrow ce_2$ *then* $\vdash ce_2 : ct$.

Proof. Only rule $\text{(CTX)}\dfrac{ce_1 \underset{\sigma}{\rightarrow} ce_2}{\mathcal{CE}^{\mathrm{f}}[\![ce_1]\!] \rightarrow \mathcal{CE}^{\mathrm{f}}[\![ce_2]\!]}\ \sigma = \mathsf{env}(ce_1, \mathcal{CE}^{\mathrm{f}})$

reduces a top level class expression. By Lemma 5 we know that ce_1 is well-typed. We get the thesis by Lemma 9 and Lemma 8. □

Soundness w.r.t. (expression) reduction follows from progress and subject reduction theorems below.

Theorem 11 (Progress)**.** *If* $\vdash \sigma : \Delta$*,* $\mathsf{isComplete}(\Delta)$*, and* $\Delta; \emptyset \vdash e : c$ *then either* e *is a value or* $e \underset{\sigma}{\rightarrow} \_$.

Proof. By induction over the expression typing rules. We show only one case:

Case $\text{(INVK-T)}\dfrac{\begin{array}{c}\Delta; \emptyset \vdash e : c \\ \Delta; \emptyset \vdash \overline{e} : \overline{c} \\ \Delta \vdash \overline{c} \leq (\overline{c}'[\mathsf{from}\ c])\end{array}}{\Delta; \emptyset \vdash e.m\,(\overline{e}) : c[\mathsf{from}\ c]}\ \mathsf{mType}(\Delta, c, m) = \overline{c}' \rightarrow c$

Assume $\overline{e} = e_1 \ldots e_n$. From the premises by inductive hypothesis we have two cases:

- $e \underset{\sigma}{\rightarrow} \_$, or $e_i \underset{\sigma}{\rightarrow} \_$ for some $i \in 1..n$. We can apply (CTX) to reduce the whole term.
- e is a value of form $c\,(\overline{fv})$, and $\overline{e} = \overline{v}$. Hence, we have applied typing rule

$$\text{(OBJ-T)}\ \frac{\begin{array}{c}\Delta;\emptyset \vdash v_i : c_i \quad \forall i \in 1..n \\ \Delta \vdash c_i \leq (c_i'[\mathsf{from}\ c]) \quad \forall i \in 1..n\end{array}}{\Delta;\emptyset \vdash c\,(\overline{fv}) : c}\ \begin{array}{l}\mathsf{nonAbs}(\Delta, c) \\ \overline{fv} = \mathtt{this}.f_1 = v_1\mathtt{;}\ \ldots\ \mathtt{this}.f_n = v_n\mathtt{;} \\ \mathsf{defFields}(\Delta, c) = f_1{:}c_1', \ldots, f_n{:}c_n'\end{array}$$

  We can apply rule

$$\text{(INVK)}\ \frac{}{c\,(\overline{fv}).m\,(\overline{v}) \ \underset{\sigma}{\rightarrow}\ e[\mathsf{from}\ c][\overline{v}/_{\overline{x}}][^{c\,(\overline{fv})}/_{\mathtt{this}}]}\ \mathsf{mBody}(\sigma, c, m) = \langle \overline{x}, e \rangle$$

  Indeed:
  - $\mathsf{mBody}(\sigma, c, m) = \langle \overline{x}, e \rangle$ holds by $\mathsf{mType}(\Delta, c, m) = \overline{c} \rightarrow c$ and $\mathsf{nonAbs}(\Delta, c)$,
  - $|\ \overline{v}\ | = |\ \overline{x}\ |$ holds, since by second premise we have $|\ \overline{v}\ | = |\ \overline{c}\ |$, by $\Delta \vdash \overline{c} \leq (\overline{c}'[\mathsf{from}\ c])$ we have $|\ \overline{c}\ | = |\ \overline{c}'\ |$, and by (BASIC-T), (METHOD-T) and definition of mType we have $|\ \overline{x}\ | = |\ \overline{c}'\ |$. □

In order to prove subject reduction w.r.t. (expression) reduction we state the following interesting lemmas.

Lemma 12.

1. $c_1[\mathsf{from}\ \mathtt{outer}.c_2] = c_1'[\mathsf{from}\ \mathtt{outer}.c_2']$ *iff* $c_1'[\mathsf{from}\ c_2'] = c_1'[\mathsf{from}\ c_2']$
2. $c_1.C[\mathsf{from}\ c_2] = c_1'.C[\mathsf{from}\ c_2']$ *iff* $c_1[\mathsf{from}\ c_2] = c_1'[\mathsf{from}\ c_2']$

Proof. The thesis trivially holds by definition of $\_[\mathsf{from}\ \_]$. □

Lemma 13. $c[\mathsf{from}\ c_1][\mathsf{from}\ c_2] = c[\mathsf{from}\ c_1[\mathsf{from}\ c_2]]$

Proof. Let us denote by π^n a path of length n, and by π^{n-j}, well-formed only if $0 \leq j \leq n$, and the path composed by the first $n - j$ elements of π^n.

By definition of $\_[\mathsf{from}\ \_]$ we have that
$\mathtt{outer}^n.\pi^k[\mathsf{from}\ \mathtt{outer}^{n_1}.\pi^{k_1}][\mathsf{from}\ \mathtt{outer}^{n_2}.\pi^{k_2}] = (\mathtt{outer}^{n_1}.(\pi^{k_1} \backslash n).\pi^k)[\mathsf{from}\ \mathtt{outer}^{n_2}.\pi^{k_2}]$
and
$\mathtt{outer}^n.\pi^k[\mathsf{from}\ \mathtt{outer}^{n_1}.\pi^{k_1}[\mathsf{from}\ \mathtt{outer}^{n_2}.\pi^{k_2}]] = \mathtt{outer}^n.\pi^k[\mathsf{from}\ \mathtt{outer}^{n_2}.(\pi^{k_2} \backslash n_1).\pi^{k_1}]$.
By Lemma 12, we have to show only that:
$(\mathtt{outer}^{n_1}.(\pi^{k_1} \setminus n))[\mathsf{from}\ \pi^{k_2}] = \mathtt{outer}^n[\mathsf{from}\ (\pi^{k_2} \setminus n_1).\pi^{k_1}]$
By cases:

$n \geq k_1$ **and** $n_1 \geq k_2$
$\mathtt{outer}^{n_1+n-k_1}[\mathsf{from}\ \pi^{k_2}] = \mathtt{outer}^n[\mathsf{from}\ \mathtt{outer}^{n_1-k_2}.\pi^{k_1}]$
$\pi^{k_2} \setminus n_1 + n - k_1 = \mathtt{outer}^{n_1-k_2}.(\pi^{k_1} \setminus n)$
$\mathtt{outer}^{n_1+n-k_1-k_2} = \mathtt{outer}^{n_1-k_2+n-k_1}$

$n < k_1$ **and** $n_1 \geq k_2$
$(\mathtt{outer}^{n_1}.\pi^{k_1-n})[\mathsf{from}\ \pi^{k_2}] = \mathtt{outer}^n[\mathsf{from}\ \mathtt{outer}^{n_1-k_2}.\pi^{k_1}]$
$(\pi^{k_2} \setminus n_1).\pi^{k_1-n} = \mathtt{outer}^{n_1-k_2}.(\pi^{k_1} \setminus n)$
$\mathtt{outer}^{n_1-k_2}.\pi^{k_1-n} = \mathtt{outer}^{n_1-k_2}.\pi^{k_1-n}$

$n \geq k_1$ **and** $n_1 < k_2$
$\mathtt{outer}^{n_1+n-k_1}[\mathsf{from}\ \pi^{k_2}] = \mathtt{outer}^n[\mathsf{from}\ \pi^{k_2-n_1}.\pi^{k_1}]$
$\pi^{k_2} \setminus n_1 + n - k_1 = (\pi^{k_2-n_1}.\pi^{k_1}) \setminus n$
$\pi^{k_2-n_1} \setminus n - k_1 = \pi^{k_2-n_1} \setminus n - k_1$

$n < k_1$ **and** $n_1 < k_2$
$\texttt{outer}^{n_1}.\pi^{k_1-n}[\mathsf{from}\ \pi^{k_2}] = \texttt{outer}^{n}[\mathsf{from}\ \pi^{k_2-n_1}.\pi^{k_1}]$
$(\pi^{k_2} \setminus n_1).\pi^{k_1-n} = (\pi^{k_2-n_1}.\pi^{k_1}) \setminus n$
$\pi^{k_2-n_1}.\pi^{k_1-n} = \pi^{k_2-n_1}.\pi^{k_1-n}$ □

The following lemma states that the type of an expression is preserved (modulo moving paths) when the expression is copied into a new position.

Lemma 14 (Moving paths preserves typing)**.** *If* $\Delta[\mathsf{in}\ c]; \Gamma \vdash e : c$ *then* $\Delta; \Gamma[\mathsf{from}\ c] \vdash e[\mathsf{from}\ c] : c[\mathsf{from}\ c]$.

Proof. By induction over the typing rules. We show only one case, the others are either similar or trivial.

(FIELD-ACCESS-T) We have

(a) $\Delta[\mathsf{in}\ c]; \Gamma \vdash e : c'$ from the premise,

(b) $\mathsf{mType}(\Delta[\mathsf{in}\ c], c', f) = c$ from the side condition, and

(c) $\Delta; \Gamma[\mathsf{from}\ c] \vdash e[\mathsf{from}\ c] : c'[\mathsf{from}\ c]$ from the inductive hypothesis and (a).

We have to prove that
$\Delta; \Gamma[\mathsf{from}\ c] \vdash e.f[\mathsf{from}\ c] : c[\mathsf{from}\ c'][\mathsf{from}\ c]$.
By (b), (BASIC-T) and definition of _[in _] we know that
$\mathsf{mType}(\Delta, c'[\mathsf{from}\ c], f) = c$.
By definition $e.f[\mathsf{from}\ c] = e[\mathsf{from}\ c].f$, which is typed by
rule (FIELD-ACCESS-T), with premise (c), in this way:
$\Delta; \Gamma[\mathsf{from}\ c] \vdash e[\mathsf{from}\ c].f : c[\mathsf{from}\ c'[\mathsf{from}\ c]]$.
Finally, by Lemma 13 we have that $c[\mathsf{from}\ c'[\mathsf{from}\ c]] = c[\mathsf{from}\ c'][\mathsf{from}\ c]$. □

Lemma 15 (Moving paths preserves subtyping)**.** *If* $\Delta[\mathsf{in}\ c] \vdash c_1 \leq c_2$ *then* $\Delta \vdash c_1[\mathsf{from}\ c] \leq c_2[\mathsf{from}\ c]$

Proof. Analogous to the previous lemma. □

Lemma 16 (Substitution)**.** *If* $\Delta; \Gamma, x{:}c_1 \vdash e : c_1'$, $\Delta; \emptyset \vdash v : c_2$ *and* $\Delta \vdash c_2 \leq c_1$, *then* $\Delta; \Gamma \vdash e[^{v}/_{x}] : c_2'$ *and* $\Delta \vdash c_2' \leq c_1'$.

Lemma 17 ($\mathcal{E}$ substitution)**.** *If* $\Delta \vdash c_2 \leq c_1$ *then:*

1. *If* $\Delta; \emptyset \vdash e_1 : c_1$, $\Delta; \emptyset \vdash e_2 : c_2$, *and* $\Delta; \emptyset \vdash \mathcal{E}^{\mathrm{r}}[\![e_1]\!] : c_1'$ *then* $\Delta; \emptyset \vdash \mathcal{E}^{\mathrm{r}}[\![e_2]\!] : c_2'$ *and* $\Delta \vdash c_2' \leq c_1'$.

2. *If* $\Delta; \Gamma, \overline{x}{:}\overline{c}_1 \vdash e : c_1$, $\Delta \vdash \overline{c}_1 \leq \overline{c}_2$. $\Delta; \emptyset \vdash \overline{e} : \overline{c}_2$ *then* $\Delta; \Gamma \vdash e[^{\overline{e}}/_{\overline{x}}] : c_2$ *and* $\Delta \vdash c_1 \leq c_2$.

Theorem 18 (Subject reduction)**.** *If* $\vdash \sigma : \Delta$, $\mathsf{isComplete}(\Delta)$, $\Delta; \Gamma \vdash e_1 : c_1$ *and* $e_1 \xrightarrow[\sigma]{} e_2$, *then* $\Delta; \Gamma \vdash e_2 : c_2$ *and* $\Delta \vdash c_2 \leq c_1$.

Proof. By induction over the reduction rules. We show only one case, the others are either similar or trivial.

Case $\dfrac{}{c(\overline{fv}).m(\overline{v}) \xrightarrow[\sigma]{} e[\mathsf{from}\ c][^{\overline{v}}/_{\overline{x}}][^{c(\overline{fv})}/_{\texttt{this}}]}\ (\textsc{invk})\ \mathsf{mBody}(\sigma, c, m) = \langle \overline{x}, e \rangle$

typed by $\dfrac{\begin{array}{c}\Delta;\Gamma \vdash c\,(\overline{fv})\,:\,c \\ \Delta;\Gamma \vdash \overline{v} : \overline{c} \\ \Delta \vdash \overline{c} \leq (\overline{c}'[\mathsf{from}\ c])\end{array}}{\Delta;\Gamma \vdash c\,(\overline{fv}).m\,(\overline{v})\,:\,c_0[\mathsf{from}\ c]}\ (\text{INVK-T})\ \begin{array}{l}\mathsf{mType}(\Delta, c, m) = \overline{c}' \to c_0 \\ \overline{c}' = c_1 \ldots c_n\end{array}$

In this case, $c\,(\overline{fv})$ is typed by rule

$$\dfrac{\begin{array}{c}\Delta;\Gamma \vdash v_i : c_i \quad \forall i \in 1..n \\ \Delta \vdash c_i \leq (c_i'[\mathsf{from}\ c]) \quad \forall i \in 1..n\end{array}}{\Delta;\Gamma \vdash c\,(\overline{fv})\,:\,c}\ (\text{OBJ-T})\ \begin{array}{l}\mathsf{nonAbs}(\Delta, c) \\ \overline{fv} = \mathtt{this}.f_1 = v_1\mathtt{;}\ \ldots\ \mathtt{this}.f_n = v_n\mathtt{;} \\ \mathsf{defFields}(\Delta, c) = f_1 : c_1', \ldots, f_n : c_n'\end{array}$$

By definition of $\mathsf{nonAbs}(\Delta, c)$ and $\mathsf{isComplete}(ct)$, we know that all methods of c are not **`abstract`**. By definition of $\mathsf{mType}(\Delta, c, m)$ we know that $\mathsf{cType}(\Delta, c)$ contains a (non abstract) method m with body e.
Since $\vdash \sigma : \Delta$, all classes in σ are typed by rule (BASIC-T). So we know that

$$\dfrac{\Delta[\mathsf{in}\ c];\Gamma \vdash e : c \quad \Delta[\mathsf{in}\ c] \vdash c \leq c_0}{\Delta[\mathsf{in}\ c] \vdash mh\{\ \mathtt{return}\ e\mathtt{;}\ \} : (m{:}c_1 \ldots c_n \to c_0)}\ (\text{METHOD-T})\ \begin{array}{l}\Gamma = \mathtt{this}{:}\Lambda, x_1{:}c_1, \ldots, x_n{:}c_n \\ mh = c_0\ m(c_1\ x_1, \ldots, c_n\ x_n) \\ \mathsf{exists}(\Delta, c_i) \quad \forall i \in 0..n\end{array}$$

holds. By Lemma 14 $\Delta;\Gamma[\mathsf{from}\ c] \vdash e[\mathsf{from}\ c] : c[\mathsf{from}\ c]$ holds.
By Lemma 17-(2) we know that $\Delta;\Gamma[\mathsf{from}\ c] \vdash e[\mathsf{from}\ c][\overline{v}/\overline{x}][^{c\,(\overline{fv})}/_{\mathtt{this}}] : c'$ and $\Delta \vdash c' \leq c[\mathsf{from}\ c]$. From $\Delta[\mathsf{in}\ c] \vdash c \leq c_0$, by Lemma 15 we know that
$\Delta \vdash c[\mathsf{from}\ c] \leq c_0[\mathsf{from}\ c]$, and the proof is concluded by subtyping rule (TRANS-S). □