# Jam - A Smooth Extension of Java with Mixins*

Davide Ancona, Giovanni Lagorio, and Elena Zucca

Dipartimento di Informatica e Scienze dell'Informazione
Via Dodecaneso, 35,16146 Genova (Italy)
email: davide,zucca@disi.unige.it
fax: +39 010-3536699

**Abstract.** In this paper we present Jam, an extension of the Java language supporting *mixins*, that is, parametric heir classes. A mixin declaration in Jam is similar to a Java heir class declaration, apart that it does not extend a fixed parent class, but simply specifies the set of fields and methods a generic parent should provide. In this way, the same mixin can be instantiated on many parent classes, producing different heirs, thus avoiding code duplication and largely improving modularity and reuse. Moreover, as happens for classes and interfaces, mixin names are reference types, and all the classes obtained instantiating the same mixin are considered subtypes of the corresponding type, hence can be handled in a uniform way through the common interface. This possibility allows a programming style where different ingredients are "mixed" together in defining a class; this paradigm is partly similar to that based on multiple inheritance, but avoids its complication.
The language has been designed with the main objective in mind to obtain, rather than a new theoretical language, a working and smooth extension of Java. That means, on the design side, that we have faced the challenging problem of integrating the Java overall principles and complex type system with this new notion; on the implementation side, that we have developed a Jam to Java translator which makes Jam sources executable on every Java Virtual Machine.

## 1 Introduction

In the last years, the notion of *parametric heir class* or *mixin* (following the terminology originally introduced in [17,16]) has deserved great interest in the programming languages community. As the first name suggests, a mixin is a uniform extension of many different parent classes with the same set of fields and methods, that is, a class-to-class function. To be more concrete, let us consider a schematic class declaration in Java.

```
class H1 extends P1 { decs }
```

where `P1` is some parent class and *decs* denotes a set of field and method declarations. In Java, as in most other object-oriented programming languages, if we want to extend another parent class, say `P2`, with *the same* set of fields and methods, then we have to write a new independent declaration, duplicating the code in *decs*.

```
class H2 extends P2 { decs }
```

Assume now to have a language allowing to give a name, say `M`, to *decs*, and to instantiate `M` on different parent classes, e.g. `P1` and `P2`, obtaining different heir classes equivalent to `H1` and `H2` above.

```
mixin M { decs }
class H1 = M extends P1 ;
class H2 = M extends P2 ;
```

* Partially supported by Murst - Tecniche formali per la specifica, l'analisi, la verifica, la sintesi e la trasformazione di sistemi software

Then we say that `M` is a *mixin*.

A mixin declaration resembles a usual heir class declaration, apart that a mixin does not refer to a fixed parent class, but simply specifies the set of fields and methods a generic parent should provide. The fact that the same mixin can be instantiated on many parent classes avoids code duplication and largely improves modularity and reuse. The name refers to the fact that in a language supporting mixins it is possible to "mix", in some sense, different ingredients during class creation, as nicely illustrated through the jigsaw puzzle metaphor in [6]. This paradigm is partly similar to that based on multiple inheritance, but avoids its complication.

Mixin-based programming has been now extensively studied both on the methodological and foundational point of view [7,6,8,4,2,3]. The results can be summarized as follows. First, the mixin notion is not strictly related to object-oriented programming but can be formulated in general in the context of module composition (a *mixin module* is a module where some components are not defined but expected to be provided by some other module). This notion allows to have a clean and unifying view of different linguistic mechanisms for composing modules. Finally, the intuitive understanding of a mixin as a class-to-class function (or, in the general case, module-to-module function) can be actually supported by a rigorous mathematical model [2,3].

Despite of this advanced state of the art, few attempts have been made at designing real programming languages supporting mixins. As already mentioned, the first use of the word mixin as a technical term originates with the LISP community [16,19]. After that, at our knowledge, there exist only a proposal for extending ML [13], a working extension for Smalltalk [9] and a proposal for a Java-like mixin language [14] (whose relation with our work will be discussed in detail in Sect.5.2).

In this paper, we present Jam[1], a working and smooth extension of Java with mixins. By these two adjectives we mean that our main aim is to produce an executable and minimal extension of Java, rather than define a new theoretical language supporting mixins. More precisely, Jam is an upward-compatible extension of Java 1.0 (apart from two new keywords), a great effort has been spent in integrating mixin-related features with the Java overall design principles, the type system is a natural extension of the Java type system with a new kind of types (mixin types), the dynamic semantics is directly defined by translation into Java and, finally, this translation has been implemented by a Jam to Java translator which makes Jam immediately executable on every Java Virtual Machine.

The structure of the presentation is as follows. In Sect.2 we provide a user introduction to Jam, through some examples, and illustrate and motivate in detail our design choices. In Sect.3 we formally define the language, giving (a part of) the abstract syntax and the static semantics (the full definition can be find in [1]). The Jam type system is defined as a conservative extension of the Java type system. For what concerns the Java part, we basically follow the type system proved sound in [12], even though we cover some more features and take a somewhat different style of presentation. In Sect.4 we define a formal translation from Jam into Java and state the correctness of this translation w.r.t. static semantics (that is, correct Jam programs are expanded into correct Java programs; this also ensures the soundness of the Jam type system). Finally in Sect.5.1 we briefly describe the implementation, provide a detailed comparison with the proposals in [9,14] and outline further research directions.

An extended version of this paper, including the full type system, the proof of correctness of the translation and more examples and discussions, is [1].

The Jam compiler and the sources are available at: `http://gio.libertyline.com/jam`.

[1] **J**ava + **m**ixin = **jam**

## 2 User Introduction and Rationale

In this section we provide a user introduction to Jam and illustrate and motivate our design choices. In 2.1 we give an overall view of the capabilities added to Java by the introduction of mixins, in 2.2-2.5 we discuss some more specific points, and finally in 2.6 we point out the main limitations of the language.

### 2.1 An example

Fig. 1 shows the declaration of the mixin `Undo`. We use `typewriter` style for code fragments. This mixin, as the name suggests, provides an "undo" mechanism that permits to restore the

```
mixin Undo {
 inherited String getText() ;
 inherited void setText(String s) ;
 String lastText ;
 void setText(String s) {
  lastText = getText() ;
  super.setText(s) ;
 }
 void undo() {
  setText(lastText) ;
 }
}
```

**Fig. 1.** Mixin declaration

text before the last modification. As shown in the example, a mixin declaration is logically split in two parts: the declarations of the components which are expected to be provided by the parent class, prefixed by the `inherited` modifier, and the declarations of the components defined in the mixin. Note that defined components can override/hide inherited components, as it happens for usual heir classes.

The mixin `Undo` can be instantiated on classes that define two non-abstract methods `getText` and `setText`, with types as specified in the `inherited` declaration. Fig. 2 shows

```
class Textbox extends Component {
 String text ;
 ...
 String getText() { ... }
 void setText(String s) { ... }
}

class TextboxWithUndo = Undo extends Textbox {}
```

**Fig. 2.** Mixin instantiation

an example of instantiation; we have used as parent a class `Textbox` which extends a generic class `Component`. In the instantiation no constructors are specified for the new class

TextboxWithUndo (they should be declared between the curly braces) and so, as in Java, it is assumed that the class has only the default constructor. To obtain a correct instantiation Textbox must define the mixin inherited part by implementing the methods getText and setText. These methods must have the same return and arguments type and equivalent[2] throws clause w.r.t. the corresponding inherited declaration. The classes obtained by instantiating the mixin provide, in addition to the methods getText (inherited from parent class) and setText (inherited and overridden), all other fields and methods of the class Textbox, the method undo and the field lastText.

The expected semantics of mixin instantiation can be informally expressed by the following *copy principle*:

> A class obtained instantiating a mixin $M$ on a parent class $P$ should have the same behavior as a usual heir of $P$ whose body contains a copy of all the components defined in $M$.

A class implementing the mixin inherited part can nevertheless be an invalid parent for instantiation, since there is another requirement to be met: the heir class obtained instantiating the mixin must be a correct Java heir class. This leads to a set of constraints which are described in detail in Sect. 2.3.

What we have seen so far shows the use of a mixin declaration as a *scheme*, that is, a parametric heir class that can be instantiated on different classes. In this way we avoid code duplication, a good result in itself, but Jam allows something more: a mixin can be used as a *type* and a mixin instance[3] is a subtype of both the mixin and the parent class on which it has been instantiated.

This allows the programmer to manipulate objects of any mixin instance by using the common interface specified by the mixin declaration (see Fig. 3). An important consequence

```
class TextboxWithUndo = Undo extends Textbox {}
class BreakIteratorWithUndo =
     Undo extends java.text.BreakIterator {}

class TestUndo {
 void f() {
  g( new TextboxWithUndo() ) ;
  g( new BreakIteratorWithUndo() ) ;
 }
 void g(Undo u) {
  u.setText("foo") ;
  u.setText("bar") ;
  System.out.println("Previous text: "+u.lastText) ;
  System.out.println("Current text : "+u.getText());
 }
}
```

**Fig. 3.** Use of mixin types

is that Jam supports a programming style (sometimes called mixin-based [7]) where different

[2] That is, every exception declared in one clause must be a subtype of an exception declared in the other, and conversely.

[3] We will call *mixin instance* a class obtained instantiating a mixin, to be not confused with an *instance* of a class.

ingredients are "mixed" together in defining a class. This paradigm has been advocated [6,5] on the methodological side since it allows to partly recover the expressive power of multiple inheritance without introducing its complication; however the novelty of Jam is that mixin-based programming is rigorously introduced in the context of a strongly typed language.

### 2.2 Other components of a mixin declaration

In the simple example presented in the previous section we have not included *all* the kinds of components which can appear in a mixin declaration.

Indeed, following the design principle that a mixin should be as similar as possible to a usual heir class, mixins should provide all their features. In the sequel we illustrate each of them in detail highlighting and justifying some restrictions.

*Interfaces* A mixin can implement an interface in exactly the same way a class does.

*Constructors* A constructor invocation in Java takes place in three cases: in an object creation expression `new C( ... )`, inside another constructor of the same class via `this` and inside an heir's constructor via `super`. However, creating objects which are instances of mixins makes no sense. Moreover, in Jam mixin instances are *not* considered heirs of the corresponding mixin. Hence, for mixins the invocation inside an heir's constructor never occurs. In summary, it makes no sense to declare constructors in mixins; however, it is possible to declare constructors for each particular mixin instance at the point of instantiation.

*Inherited instance fields* In a mixin it is possible to access inherited (instance) fields in the same way as a usual heir class does: using the field name `id` or the forms `this.id` and `super.id` (the latter is needed when a defined field hides an inherited one).

*Static members* Although in Jam static components are declared in the same way as instance components except, of course, the use of the `static` modifier, their visibility is different: they are not considered part of the mixin type. Consider, for example, the following code fragment:

```
mixin M {
 static void m() {}
 static int f ;
}
```

We do not allow in Jam invocations `M.m()` or *e*`.m()` with *e* of type `M`. However, for each class `H` obtained instantiating `M`, invocations `H.m()` or *e*`.m()`[4] with *e* of type `H` are legal. The same rule holds for fields. In other words, every class that is an instance of `M` has "its own copy" of static components declared in the mixin. Other choices are technically possible:

- sharing only one copy of the static components declared in the mixin between all mixin instances; in this case it should be allowed accessing static members through the mixin type too;
- leave to the programmer (introducing a new keyword, or analogous mechanisms) the decision whether a component should be shared between all the mixin instances or not.

In Jam, we have chosen the "unshared" version because, in this way, a mixin instantiation on a parent class is equivalent to that obtained by copying the mixin body in the declaration of the new class, as requested by the copy principle. Static components can be inherited (of course, they are not part of mixin type either) but, like in Java, static methods cannot be abstract.

[4] We maintain this alternative syntax for compatibility reasons only, see 15.10.1 of [15]

### 2.3 Constrains on instantiation

As mentioned in Sect. 2.1, the fact that a class `P` provides an implementation for the `inherited` part of a mixin `M` is not enough for ensuring that `P` can be correctly used as a parent for `M`. Indeed, in addition to methods declared `inherited` in `M`, the class `P` can contain some *other* methods which could interfere, in various ways, with methods in `M`. Let us briefly illustrate the different interference cases.

*Unexpected overriding/hiding* A method in `P` is incidentally overridden (hidden) by a method defined in `M` if it has the same name, arguments type, return type, kind (`instance` or `static`) and a compatible[5] `throws` clause. For instance, instantiating the mixin `Undo` on a class with a `void undo()` method produces an unexpected overriding. This situation looks somewhat undesirable, since there is some overriding which was not planned when declaring the mixin; however, our choice for Jam has been to consider legal these instantiations, leaving to the programmer the care of avoiding them when the additional overriding is undesired. Indeed, different choices would sensibly complicate either the static (if the choice is to forbid) or dynamic (if the choice is to keep both versions) semantics, while ours is the natural extension to mixins of what happens for usual heir classes. See Sect.5.2 for some further discussion on this point.

*Illegal overriding/hiding* A method in `P` is illegally overridden (hidden)[6] by a defined method in `M` if it has the same signature (name and arguments type) but either different return type, or different kind or incompatible `throws` clause. This is not correct in Jam.

*Ambiguous overloading* There exist contexts in which the presence of the method in `P` makes ambiguous, w.r.t. overloading resolution, an invocation of the method in `M`. Let us clarify this case with an example. Assume that the method `Undo.undo` contains the call `setText(null)`; this invocation is statically correct. Suppose now to instantiate `Undo` on a class `Boom` which defines, besides the methods `String getText()` and `void setText(String)`, the method `void setText(Integer)`. In this case the call `setText(null)` becomes ambiguous. Indeed, `null` can be implicitly converted to any reference type, hence both methods are applicable and neither is more specific[7].

In general, if two methods have the same name, then the addition of one may make ambiguous, w.r.t. overloading resolution, an invocation of the other if and only if they have the same number and type of arguments except for some argument for which they have two different reference types (see Fig. 13 in Sect.3 for the formal definition). In alternative we could have defined less strict rules by forbidding the instantiation only when some method body in the mixin contains a method invocation that would become ambiguous (as in the example). However, we have preferred to follow the principle that the correctness of a mixin instantiation should depend only on the mixin type and not on its implementation. In this way, indeed, a modification of the method bodies does not affect the correctness of the instantiation. Even though this approach has the drawback of forbidding also "good" instantiations, on the methodological side it seems more consistent with the choice of describing the requirements on the parent class via the `inherited` declarations.

### 2.4 Overloading

The Java rules for overloading resolution[8] smoothly extends to Jam, just including mixin types among other reference types and taking into account in the definition of "more specific" the fact that every mixin instance is a subtype of (hence, can be converted to) the corresponding mixin type. However, some particular care is needed for handling the situation when there is an overloading conflict between an inherited and a defined method in a mixin. Let us illustrate the problem on the following simple example.

[5] See 8.4.4 in [15]
[6] See 8.4.6.3 in [15]
[7] See 15.11.2 in [15]
[8] See 15.11.2 in [15]

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

class Parent {
 void f(B b) {}
}

class Heir extends Parent {
 void f(A a) {}
}
```

```
mixin M {
 inherited void f(B b) ;
 void f(A a) {}
}

class Test {
 void test(Heir h, B b, M m) {
  h.f(b) ; // ambiguous
  m.f(b) ; // ambiguous?
 }
}
```

**Fig. 4.** Overloading conflict between inherited and defined methods

In the first part of the code shown in Fig.4, `B` is a subtype of `A` and `Heir` is a subtype of `Parent`. The class `Parent` defines a method named `f` with one argument of type `B`, while its subclass `Heir` defines a method with the same name and argument's type `A`. Due to the symmetry of the situation, the invocation `h.f(b)`, where `h` and `b` are of type `Heir` and `B`, respectively, is ambiguous, since there are two applicable methods and neither is more specific[9].

If we consider now the declaration of the mixin `M`, the situation is exactly analogous to the preceding: a (parametric) heir class defines a method whose argument type is a supertype of the argument type of a method with the same name in the parent class. Hence, we expect the invocation of `m.f(b)`, where `m` has type `M`, to be ambiguous as well.

For achieving this goal, we assume that inherited methods in a mixin `M` are annotated with a type (that is, considered to be have been declared within the corresponding module; see [1] for the precise formal definition of annotations) which is not `M` but a special type *Parent*(`M`) which represents the generic parent on which the mixin can be instantiated, and is assumed to be a supertype of `M`.

### 2.5 Use of `this` in mixins

A last delicate point in the Jam type system concerns the use of the keyword `this`, which denotes, in an instance method (resp. constructor), the current object on which the method has been invoked (the current object to be constructed). In a method or constructor declared in a class `C`, the expression `this` has static type `C` in Java[10]. Now, we have to decide which should be the static type of `this` in a method defined in a mixin `M`. Since we want to be able to type-check the mixin declaration independently from future instantiations, the only possibility is to assume that `this` has static type `M`, since this is the only type available at mixin declaration's time. However, this is in conflict with the fact that we expect that in a class `H` instance of a mixin `M` the expression `this` has static type `H`, as it happens for usual heir classes. More precisely, the fact of having correctly type-checked the mixin declaration under the assumption that `this` has type `M` does not guarantee that (the Java class `H` corresponding to) a mixin instance (following the copy principle) is always a correct Java class, since in Java `this` has type `H` in this class. This can lead to unsound situations in some subtle cases involving overloading. Let us consider the example in Fig.5.

The class `A` declares two methods named `f` with argument's type a mixin `M` and an instance `H` of `M`, respectively. In the invocation of `f` inside the method `g` declared in `M`, since `this` has type `M`, the expression `A.f(this)` has type `int`, hence can be correctly assigned to the variable `i`.

[9] See 15.1.2.2 in [15]
[10] See 15.7.2. in [15]

```
class A {
 static int f(M m) { ... }
 static boolean f(H h) { ... }
}

mixin M {
 void g() {
  int i = A.f(this) ;
 }
}
```

```
class H = M extends Object {} ;

// ``Equivalent'' declaration for H
class H ... {
 void g() {
  int i = A.f(this) ; // Boom !!!
 }
}
```

**Fig. 5.** Problem in using `this` in mixins

Now, if the expected semantics of `H`, following the copy principle, is to be equivalent to the class shown in the figure where the declaration of `g` has been copied into the body, then the invocation `A.f(this)` has now type `boolean`, hence cannot be used for initializing the variable `i`.

This is a particular case of a more general problem, that is, the fact that in a mixin declaration in Jam there is no way to refer to the parametric types of either the parent or the heir class resulting from the instantiation. See the following section for more comments about that.

In order to avoid these situations, we have taken for Jam a quite drastic design decision, that is, to forbid the use of `this` as argument in method and constructor invocation inside a mixin.

### 2.6 Limitations

The main limitation of the language is that in a mixin it is not possible to refer either to the "generic" parent class to which the mixin will be applied or to the "generic" heir class obtained by instantiation. As an example, let us consider the following declarations of a parent class `P` and an heir `H`.

```
class P {
 static int counter ;
}
class H extends P {
 static int counter ;
 static void incrThat() { ++P.counter ; }
 ...
 int value ;
 public boolean equals(Object that) {
  if (that instanceof H) return ((H)that).value == value ;
  return false ;
 }
}
```

The definition of `H` cannot be "abstracted" in a mixin definition, for two reasons.

- The method `incrThat` explicitly refers to the parent class `P` since the static field `counter` of the superclass has been hidden by a declaration in `H`.
- The method `equals` uses the name of the heir class `H` which is unknown for a mixin.

More generally, a class `H` heir of `P` cannot be "abstracted" into a mixin when it contains:

– (explicit) references to types H and P;
– invocations of H/P constructors.

If H/P are only used for accessing static members, then H can be "abstracted" except for some cases involving hiding (as shown by the example). In Sect.5.3 we discuss possible solutions to this problem.

# 3 The Formal Definition

In this section we formally define the abstract syntax and the static semantics of Jam. The implemented version of Jam is an upward-compatible extension of Java 1.0 (apart the fact that `mixin` and `inherited` are keywords in Jam); however the formal definition only considers a subset of the language chosen in such a way to be minimal but sufficient for our aim, which is to analyze how the Java type system must be enriched in order to support mixin types (the soundness of this extension will be proved in the next section). Excluded features fall in two main categories: those which are orthogonal w.r.t. this aim, like multithreading, and those whose semantics can be trivially derived, like the `for` loop. In particular, we have excluded the following features: arrays, `final` and access modifiers, features related with linking native code and multithreading. We have included the following features not considered in [11]: constructors, `static` members, checked exceptions[11], `abstract` classes and methods, method invocations and field accesses via `super`.

In this paper, for lack of space, we include only a part of the abstract syntax and the type system, whose full version can be found in [1].

## 3.1 Notations

We use the `typewriter` style for terminal and *italic* for non terminal symbols. The terminals `iname`, `cname` and `mname` indicate, interface, class and mixin names respectively. A generic name is indicated by `name`. We use the following notations:

– $A^*$ to indicate a sequence of zero or more occurrences of A,
– $A^+$ to indicate a sequence of one or more occurrences of A,
– $[A]$ to indicate that A is optional,
– $A^{\circledast}$ to indicate a set of occurrences of A, that is, a sequence in which there are no repetitions and the order is immaterial,
– $A^{\oplus}$ to indicate a non empty set of occurrences of A.

## 3.2 Abstract syntax

Fig. 6 shows a part of the Jam abstract syntax; the LALR grammar used in the implementation can be found in the [1].

The only Jam specific productions are the first three in the figure.

In Jam an alternative way to define a (possibly `abstract`) class is to instantiate a mixin on an existing class, specifying the constructors of the new class.

A mixin declaration logically consists of two parts: the former contains the declarations of the defined components, while the latter contains the inherited components declarations, that is, the declarations of the components that should be provided by the parent class on which the mixin will be instantiated. These components are labelled with the `inherited` modifier. Moreover, the set of the implemented interfaces is specified.

## 3.3 Types

In Fig. 7 are defined the Jam types. A generic *type* can be a reference type, a primitive

[11] Checked exceptions have been considered in a recent improved version [12].

$$
\begin{array}{rcl}
\mathit{ref\text{-}type} & ::= & \texttt{mname} \\
\mathit{cdecl} & ::= & [\ \texttt{abstract}\ ]\ \texttt{class cname}\ =\ \texttt{mname extends} \\
 & & \texttt{cname}\ \{\mathit{constructor}^{\circledast}\} \\
\mathit{mdecl} & ::= & \texttt{mixin name implements iname}^{\circledast} \\
 & & \{\ \langle\ [\texttt{inherited}]\ \mathit{field}\ \rangle^{\circledast}\ \langle\ [\texttt{inherited}]\ \mathit{cmeth}\ \rangle^{\circledast}\ \} \\
 & & \\
\mathit{prog} & ::= & \mathit{decl}^{\circledast} \\
\mathit{decl} & ::= & \mathit{idecl} \mid \mathit{cdecl} \mid \mathit{mdecl} \\
\mathit{simple\text{-}type} & ::= & \mathit{prim\text{-}type} \mid \mathit{ref\text{-}type} \\
\mathit{ref\text{-}type} & ::= & \texttt{iname} \mid \texttt{cname} \\
\mathit{prim\text{-}type} & ::= & \texttt{int} \mid \texttt{boolean} \\
\mathit{ret\text{-}type} & ::= & \mathit{simple\text{-}type} \mid \texttt{void} \\
\mathit{exc\text{-}type} & ::= & \texttt{cname}^{\circledast} \\
\mathit{cdecl} & ::= & [\ \texttt{abstract}\ ]\ \texttt{class cname extends cname} \\
 & & \texttt{implements iname}^{\circledast} \\
 & & \{\ \mathit{constructor}^{\circledast}\ \mathit{field}^{\circledast}\ \mathit{cmeth}^{\circledast}\ \} \\
\mathit{idecl} & ::= & \texttt{interface iname extends iname}^{\circledast}\ \{\ \mathit{imeth}^{\circledast}\ \} \\
\mathit{imeth} & ::= & \texttt{abstract}\ \mathit{ret\text{-}type}\ \texttt{name}\ \mathit{params}\ \texttt{throws}\ \mathit{exc\text{-}type}\ ; \\
\mathit{params} & ::= & (\langle\ \mathit{simple\text{-}type}\ \texttt{name}\ \rangle^{*}) \\
\mathit{constructor} & ::= & \texttt{cname}\ \mathit{params}\ \texttt{throws}\ \mathit{exc\text{-}type} \\
 & & \{\ \texttt{super}(\mathit{expr}^{*})\ ;\ \mathit{stmts}\ \} \\
\mathit{cmeth} & ::= & [\ \texttt{static}\ ]\ \mathit{ret\text{-}type}\ \texttt{name}\ \mathit{params} \\
 & & \texttt{throws}\ \mathit{exc\text{-}type}\ \mathit{mbody} \mid \\
 & & \mathit{imeth}
\end{array}
$$

**Fig. 6.** Jam abstract syntax

$$
\begin{array}{rcl}
\mathit{type} & ::= & \mathit{ref\text{-}type} \mid \mathit{prim\text{-}type} \mid \texttt{nil} \\
\mathit{field\text{-}type} & ::= & \mathit{field\text{-}kind}\ \mathit{simple\text{-}type} \\
\mathit{field\text{-}kind} & ::= & \texttt{instance} \mid \texttt{static} \\
\mathit{args\text{-}type} & ::= & \mathit{simple\text{-}type}^{*} \\
\mathit{constr\text{-}type} & ::= & \mathit{args\text{-}type}\ \texttt{throws}\ \mathit{exc\text{-}type} \\
\mathit{meth\text{-}type} & ::= & \mathit{meth\text{-}kind}\ \mathit{ret\text{-}type}\ \texttt{throws}\ \mathit{exc\text{-}type} \\
\mathit{meth\text{-}sig} & ::= & \texttt{name}, \mathit{args\text{-}type} \\
\mathit{meth\text{-}kind} & ::= & \texttt{instance} \mid \texttt{abstract} \mid \texttt{static} \\
\mathit{fields\text{-}type} & ::= & \langle \texttt{name} : \mathit{field\text{-}type} \rangle^{\circledast} \\
\mathit{meths\text{-}type} & ::= & \langle \mathit{meth\text{-}sig} : \mathit{meth\text{-}type} \rangle^{\circledast} \\
\mathit{constrs\text{-}type} & ::= & \mathit{constr\text{-}type}^{\oplus} \\
 & & \\
\mathit{module\text{-}type} & ::= & \mathit{fields\text{-}type}\ \mathit{meths\text{-}type} \\
\mathit{class\text{-}type} & ::= & \mathit{class\text{-}kind}\ \mathit{constrs\text{-}type}\ \mathit{module\text{-}type} \\
\mathit{class\text{-}kind} & ::= & \texttt{abstract} \mid \texttt{concrete} \\
\mathit{interface\text{-}type} & ::= & \mathit{meths\text{-}type} \\
\mathit{mixin\text{-}type} & ::= & \mathit{module\text{-}type}\ \texttt{inherited}\ \mathit{module\text{-}type}
\end{array}
$$

**Fig. 7.** Jam types

type (both defined in Fig. 6) or `nil` (the type of `null`). A *field-type* consists of a simple type and a (field) *kind* indicating whether the field is instance or static. The arguments type (of a method or constructor), *args-type*, is a sequence, possibly empty, of simple types. A constructor type consists of the arguments type and the set of declared exceptions (the type *exc-type* is defined in Fig. 6). A method type consists of the kind, the return type and the set of declared exceptions. A fields type is a set of fields, that is, pairs consisting of a field name and a field type. A fields type is *legal* if field names are distinct. Analogously, a methods type is a set of methods, that is, pairs consisting of a *signature* (a method name qualified by the types of the arguments) and a method type; it is *legal* when all signatures are distinct. In the following we will consider only legal fields and methods type. The type *constrs-type* is a non-empty set of constructor types. Note that a class has always at least a constructor (if it is not explicitly given the default one is assumed).

A module type consists of a set of fields and a set of methods. A class type consists of a module type, a kind and a set of constructors. An interface type consists of a set of methods (in our subset we do not consider the `final` modifier, hence an interface cannot have fields). Finally, a mixin type consists of two module types: the defined type and the inherited type, that is the expected parent type.

### 3.4 Environments

A Jam program contains both type information and information needed at runtime (that is, the method bodies). To simplify the formal definition, following the approach used in [12], we consider two components that can be extracted in a trivial way from a program: the environment $\Gamma$, that contains the type information, and the remaining part of program consisting in a set of *body declarations*, that is, constructor and method bodies of classes and mixins (fields information are contained in $\Gamma$). The syntax of these two components is given in Fig. 8. We assume that in the environment extraction process a check is performed

$$
\begin{array}{rcl}
\textit{env} & ::= & \textit{basic-type-assertion}^{\circledast} \\
\textit{basic-type-assertion} & ::= & \texttt{cname}\ \mathrm{is}_c\ \textit{class-type} \mid \\
 & & \texttt{cname}\ <^1_c\ \texttt{cname} \mid \\
 & & \texttt{name}\ \lhd^1_i\ \texttt{iname} \mid \\
 & & \texttt{iname}\ \mathrm{is}_i\ \textit{interface-type} \mid \\
 & & \texttt{iname}\ <^1_i\ \texttt{iname} \mid \\
 & & \texttt{mname}\ \mathrm{is}_m\ \textit{mixin-type} \mid \\
 & & \texttt{cname}\ \lhd_m\ \texttt{mname} \\
\textit{body-decl} & ::= & \texttt{class cname}\ \{\ \textit{constructor}^{\circledast}\ \textit{cmeth}^{\circledast}\ \} \mid \\
 & & \texttt{mixin name}\ \{\ \textit{cmeth}^{\circledast}\ \}
\end{array}
$$

**Fig. 8.** Environments and body declarations

for avoiding duplicate declarations. Hence, the static correctness of a Jam program can be expressed by the validity of the two following judgments:

$\vdash \Gamma\diamond$

$\Gamma \vdash \{BD_1, \ldots, BD_n\}\diamond_{\mathrm{Prog}}$

The former means that $\Gamma$ is a well-formed environment so that, for instance, the subclass relationship is acyclic; the latter indicates that all the body declarations are well-formed w.r.t. the type information in $\Gamma$. The validity of these two judgments is defined inductively introducing other judgments relative to subcomponents. In this paper for lack of space, we only give an outline of the judgments related to the environment.

An environment is a set of basic type assertions having the following informal meaning:

- $C$ is$_c$ $KST\ FST\ MST$ : the class $C$ declares the specified constructors ($KST$), fields ($FST$) and methods ($MST$)

- $C <^1_c C'$ : the class $C$ directly extends the class $C'$
- $T \lhd^1_i I$ : the module (either class or mixin) $T$ directly implements the interface $I$
- $I$ is$_i$ $MST$ : the interface $I$ declares the methods specified in $MST$
- $I <^1_i I'$ : the interface $I$ directly extends the interface $I'$
- $M$ is$_m$ $MODT$ `inherited` $MODT'$ : the mixin $M$ declares the defined components $MODT$ and the inherited components $MODT'$
- $C \lhd_m M$ : the class $C$ has been defined instantiating the mixin $M$

We define now some auxiliary notations used in the sequel, well-defined on environments which do not contain duplicate declarations, as we have assumed.

$$\text{Set } \Gamma(\texttt{id}) = \begin{cases} CT & \text{if } \texttt{id} \text{ is}_c\ CT \in \Gamma \\ IT & \text{if } \texttt{id} \text{ is}_i\ IT \in \Gamma \\ MXT & \text{if } \texttt{id} \text{ is}_m\ MXT \in \Gamma \\ \bot & \text{otherwise} \end{cases}$$

- $Classes(\Gamma)$ the set of all class names defined in $\Gamma$, that is, $C \in Classes(\Gamma)$ iff $C$ is$_c$ $CT \in \Gamma$,
- $Interfaces(\Gamma)$ the set of all interface names defined in $\Gamma$, that is, $I \in Interfaces(\Gamma)$ iff $I$ is$_i$ $IT \in \Gamma$,
- $Mixins(\Gamma)$ the set of all mixin names defined in $\Gamma$, that is, $M \in Mixins(\Gamma)$ iff $M$ is$_m$ $MXT \in \Gamma$.

### 3.5 Type system (outline)

In this subsection, we give the first part of the metarules of the Jam type system, that is, those related to environments. Metarules fall in two categories: those which belong to the Java type system, and those related to features introduced by Jam, which are distinguished by a label. The judgments of these metarules have generic form $\Gamma \vdash \gamma$, with $\Gamma$ an environment and $\gamma$ a *type assertion*.

*Basis* The following metarule provides the basis for the inductive definition of the validity of judgments.

$$(1) \quad \frac{}{\Gamma \vdash \gamma} \quad \gamma \in \Gamma$$

**Fig. 9.** Basic type assertion

*Relations between types* The metarules in Fig.10 all define relevant relations between reference types (that is, either classes, or interfaces, or mixins) which can be derived from the basic relations contained in the environment. In particular, the reflexive (on existing class types) and transitive closure of the relation $<^1_c$ is the *subclass* relation $\leq_c$; analogously the reflexive (on existing interface types) and transitive closure of $<^1_i$ is the *subinterface* relation $\leq_i$. The *implementation* relation from classes to interfaces is derived from $\lhd^1_i$ and the subclass and subinterface relations. The new relation introduced in Jam w.r.t. Java is that of *instantiation*, denoted $\lhd_m$, from a mixin instance to the corresponding mixin type. Finally, from all these relations we can derive a more general relation of *widening* between reference types, denoted by $\leq$.

The metarules in Fig. 11 define subtyping relations for exceptions, fields, methods and module types. These relations basically express that a module type is a subtype of another if it has more fields and/or methods; the common fields and methods must have *exactly* the same type, modulo equivalence of exceptions types ($\Gamma \vdash ET =_e ET'$ in (22) stands for $\Gamma \vdash ET \leq_e ET'$ and $\Gamma \vdash ET' \leq_e ET$). Note that, in (19), it is possible that $E_i = E_j$

$$(2)\quad \frac{\Gamma \vdash C <^1_c C'}{\Gamma \vdash C \leq_c C'} \qquad (3)\quad \frac{\Gamma \vdash C \text{ is}_c\ CT}{\Gamma \vdash C \leq_c C}$$

$$(4)\quad \frac{\Gamma \vdash C \leq_c C' \quad \Gamma \vdash C' \leq_c C''}{\Gamma \vdash C \leq_c C''} \qquad (5)\quad \frac{\Gamma \vdash I <^1_i I'}{\Gamma \vdash I \leq_i I'}$$

$$(6)\quad \frac{\Gamma \vdash I \text{ is}_i\ IT}{\Gamma \vdash I \leq_i I} \qquad (7)\quad \frac{\Gamma \vdash I \leq_i I' \quad \Gamma \vdash I' \leq_i I''}{\Gamma \vdash I \leq_i I''}$$

$$(8)\quad \frac{\Gamma \vdash T \lhd^1_i I}{\Gamma \vdash T \lhd_i I} \qquad (9)\quad \frac{\Gamma \vdash C \leq_c C' \quad \Gamma \vdash C' \lhd_i I}{\Gamma \vdash C \lhd_i I}$$

$$(10)\quad \frac{\Gamma \vdash T \lhd_i I \quad \Gamma \vdash I \leq_i I'}{\Gamma \vdash T \lhd_i I'} \qquad (11)\quad \frac{\Gamma \vdash C \leq_c C'}{\Gamma \vdash C \leq C'}$$

$$(12)\quad \frac{\Gamma \vdash I \leq_i I'}{\Gamma \vdash I \leq I'} \qquad (13\text{-Jam})\quad \frac{\Gamma \vdash C \lhd_m M}{\Gamma \vdash C \leq M}$$

$$(14)\quad \frac{\Gamma \vdash T \leq T}{\Gamma \vdash T \leq \text{Object}} \qquad (15)\quad \frac{\Gamma \vdash T \leq T}{\Gamma \vdash \text{nil} \leq T}$$

$$(16)\quad \frac{\Gamma \vdash T \lhd_i I}{\Gamma \vdash T \leq I} \qquad (17\text{-Jam})\quad \frac{\Gamma \vdash M \text{ is}_m\ MXT}{\Gamma \vdash M \leq M}$$

$$(18)\quad \frac{\Gamma \vdash T \leq T' \quad \Gamma \vdash T' \leq T''}{\Gamma \vdash T \leq T''}$$

**Fig. 10.** Subclass, subinterface, implementation and widening relations

$$(19)\quad \frac{\begin{array}{c}\Gamma \vdash E_1 \leq_c E'_1 \ldots \ \Gamma \vdash E_n \leq_c E'_n \\ \Gamma \vdash \{E_1, \ldots, E_n\} \diamond_{\text{ExcType}} \ \ \Gamma \vdash \{E'_1, \ldots, E'_n\} \diamond_{\text{ExcType}}\end{array}}{\Gamma \vdash \{E_1, \ldots, E_n\} \leq_e \{E'_1, \ldots, E'_n\}} \qquad n \geq 0$$

$$(20)\quad \frac{\Gamma \vdash T \diamond_{\text{SimpleType}}}{\Gamma \vdash K\ T \leq_{field} K\ T}$$

$$(21)\quad \frac{\Gamma \vdash \{f_1 : FT_1, \ldots, f_n : FT_n\} \diamond_{\text{FieldsType}}}{\Gamma \vdash \{f_1 : FT_1, \ldots, f_n : FT_n\} \leq_{fields} \{f_1 : FT_1, \ldots, f_m : FT_m\}} \qquad m \leq n$$

$$(22)\quad \frac{\Gamma \vdash RT \diamond_{\text{RetType}} \ \ \Gamma \vdash ET =_e ET'}{\begin{array}{l}\Gamma \vdash K\ RT\ \texttt{throws}\ ET \leq_{meth}\ K\ RT\ \texttt{throws}\ ET' \\ \Gamma \vdash \texttt{instance}\ RT\ \texttt{throws}\ ET \leq_{meth}\ \texttt{abstract}\ RT\ \texttt{throws}\ ET\end{array}}$$

$$(23)\quad \frac{\begin{array}{c}\Gamma \vdash \{msig_1 : MT_1, \ldots, msig_n : MT_n\} \diamond_{\text{MethsType}} \\ \Gamma \vdash \{msig_1 : MT'_1, \ldots, msig_k : MT'_k\} \diamond_{\text{MethsType}} \\ \Gamma \vdash MT_1 \leq_{meth} MT'_1 \ \ldots \ \Gamma \vdash MT_k \leq_{meth} MT'_k\end{array}}{\begin{array}{c}\Gamma \vdash \{msig_1 : MT_1, \ldots, msig_n : MT_n\} \leq_{meths} \\ \{msig_1 : MT'_1, \ldots, msig_k : MT'_k\}\end{array}} \qquad k \leq n$$

$$(24)\quad \frac{\Gamma \vdash FST \leq_{fields} FST' \ \Gamma \vdash MST \leq_{meths} MST'}{\Gamma \vdash FST\ MST \leq_{mod} FST'\ MST'}$$

**Fig. 11.** Relations on exceptions, fields, methods and module type

or $E'_i = E'_j$ holds for some $i, j$. Finally, the same figure show subtyping relations for field, method and module types.

We omit for lack of space the metarules defining well-formedness of Jam types.

*Type assignments* The metarules in Fig.12 define *type assignments*, that is, the fact that some Jam module (either class or interface or mixin) has a given type.

We use some auxiliary functions:

- $\mathit{ParentInterfaces}(\Gamma, I) = \{I' | I <^1_i I' \in \Gamma\}$
- $\mathit{ImplementedInterfaces}(\Gamma, T) = \{I | T \lhd^1_i I \in \Gamma\}$

Moreover, we use the auxiliary *update* operations on (*legal*) Jam fields and methods types defined below.

$$\{f_1 : FT_1, \ldots, f_n : FT_n\}[f : FT] =_{\text{def}} \{f_i : FT_i | f_i \neq f\} \cup \{f : FT\}$$

$$\{msig_1 : MT_1, \ldots, msig_n : MT_n\}[msig : MT]_\Gamma =_{\text{def}} \begin{cases} \{msig_i : MT_i | msig_i \neq msig\} \cup \{msig : MT\} \\ \text{if } \forall i \in \{1, \ldots, n\} \;\; msig = msig_i \Rightarrow \\ \qquad \begin{cases} \mathit{Kind}(MT) = \texttt{static} \Leftrightarrow \mathit{Kind}(MT_i) = \texttt{static}, \\ \Gamma \vdash \mathit{Exc}(MT) \leq_e \mathit{Exc}(MT_i), \\ \mathit{Ret}(MT) = \mathit{Ret}(MT_i) \end{cases} \\ \bot \text{ otherwise} \end{cases}$$

The functions *Kind*, *Exc* and *Ret* denote the obvious projections for methods types.

The three conditions above on updating methods types correspond to the three following Java rules on overriding:

- an instance method cannot override a static method, and conversely[12];
- a method overriding another cannot throw an exception which is not a subtype of some exception thrown by the overridden method[13];
- a method overriding another cannot have different return type[14].

It is easy to see that the update operations can be safely generalized in the obvious way to the case where the second argument is a valid fields (resp. methods) type.

On methods types we define moreover a "sum" operation $\overset{\Gamma}{\oplus}$ which is basically set union, a part that, in the case of methods with the same signature, kind, and return type, it produces just one such method whose exceptions type is the "intersection" of the exceptions types, defined below. This operation is needed in the case a class inherits (from the implemented interfaces) many methods which differ only for the `throws` clause.

$$\{msig : K\ RT\ \texttt{throws}\ ET\} \overset{\Gamma}{\oplus} \{msig' : K'\ RT'\ \texttt{throws}\ ET'\} =_{\text{def}} \begin{cases} \{msig : K\ RT\ \texttt{throws}\ ET, \\ msig' : K'\ RT'\ \texttt{throws}\ ET'\} & \text{if } msig \neq msig' \\ \{msig : K\ RT\ \texttt{throws}\ (ET \overset{\Gamma}{\otimes} ET')\} & \text{if } K = K' \wedge msig = msig' \wedge \\ & \wedge RT = RT' \\ \bot & \text{otherwise} \end{cases}$$

$$\{msig_1 : MT_1, \ldots, msig_n : MT_n\} \overset{\Gamma}{\oplus} \{msig'_1 : MT'_1, \ldots, msig'_k : MT'_k\} =_{\text{def}} \{msig_1 : MT_1\} \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} \{msig_n : MT_n\} \overset{\Gamma}{\oplus} \{msig'_1 : MT'_1\} \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} \{msig'_k : MT'_k\}$$

$$\{E_1, \ldots, E_n\} \overset{\Gamma}{\otimes} \{E'_1, \ldots, E'_m\} =_{\text{def}} \{E_i | \exists j : \Gamma \vdash E_i \leq_c E'_j\} \cup \{E'_i | \exists j : \Gamma \vdash E'_i \leq_c E_j\}$$

---

12 See 8.4.6.1 and 8.4.6.2 in [15]

13 See 8.4.6.3 in [15]

14 See 8.4.6.3 in [15]

(25)
$$\frac{\Gamma \vdash I \text{ is}_i\ MST \quad \Gamma \vdash MST \diamond_{\text{InterfaceType}} \quad \Gamma \vdash I_1 : \emptyset\ MST_1 \ldots \Gamma \vdash I_n : \emptyset\ MST_n}{\Gamma \vdash I : \emptyset\ (MST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} MST_n)[MST]_\Gamma} \qquad \begin{array}{l} n \geq 0 \\ \{I_1, \ldots, I_n\} = \mathit{ParentInterfaces}(\Gamma, I) \end{array}$$

Set:
$MXT = FST\ MST\ \texttt{inherited}\ FST'\ MST'$

(26-Jam)
$$\frac{\Gamma \vdash M \text{ is}_m\ MXT \quad \Gamma \vdash MXT \diamond_{\text{MixinType}} \quad \Gamma \vdash I_1 : \emptyset\ MST_1 \ldots \Gamma \vdash I_n : \emptyset\ MST_n}{\Gamma \vdash M :\ FST'[FST]\ ((MST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} MST_n)[MST']_\Gamma)[MST]_\Gamma} \qquad \begin{array}{l} n \geq 0 \\ \{I_1, \ldots, I_n\} = \\ \quad \mathit{ImplementedInterfaces}(\Gamma, M) \end{array}$$

(27)
$$\frac{}{\Gamma \vdash \texttt{Object} : \emptyset\ \emptyset}$$

Set:
$FST_c = FST'[FST]$
$MST_c = ((MST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} MST_n)[MST']_\Gamma)[MST]_\Gamma$
$CT = K\ KST\ FST\ MST$

(28)
$$\frac{\Gamma \vdash C \text{ is}_c\ CT \quad \Gamma \vdash CT \diamond_{\text{ClassType}} \quad \Gamma \vdash C <_c^1 C' \quad \Gamma \vdash C' : FST'\ MST' \quad \Gamma \vdash I_1 : MST_1 \ldots \Gamma \vdash I_n : MST_n}{\Gamma \vdash C : FST_c\ MST_c} \qquad \begin{array}{l} n \geq 0 \\ C \lhd_m M \notin \Gamma \\ \{I_1, \ldots, I_n\} = \mathit{ImplementedInterfaces}(\Gamma, C) \\ K = \texttt{concrete} \Rightarrow \\ \quad \mathit{Kind}(MST_c) = \texttt{concrete} \end{array}$$

Set:
$FST_c = FST'[FST_d]$
$MST_c = ((MST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} MST_n)[MST']_\Gamma)[MST_d]_\Gamma$
$MST_m = ((MST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} MST_n)[MST_i]_\Gamma)[MST_d]_\Gamma$
$CT = K\ KST\ \emptyset\ \emptyset$

(29-Jam)
$$\frac{\begin{array}{l} \Gamma \vdash C \text{ is}_c\ CT \\ \Gamma \vdash CT \diamond_{\text{ClassType}} \\ \Gamma \vdash C \lhd_m M \\ \Gamma \vdash M \text{ is}_m\ FST_d\ MST_d\ \texttt{inherited}\ FST_i\ MST_i \\ \Gamma \vdash C <_c^1 C' \\ \Gamma \vdash C' : FST'\ MST' \\ \Gamma \vdash FST'\ MST' \leq_{mod} FST_i\ MST_i \\ \Gamma \vdash I_1 : MST_1 \ldots \Gamma \vdash I_n : MST_n \end{array}}{\Gamma \vdash C : FST_c\ MST_c} \qquad \begin{array}{l} n \geq 0 \\ \{I_1, \ldots, I_n\} = \\ \quad \mathit{ImplementedInterfaces}(\Gamma, M) \\ K = \texttt{concrete} \Rightarrow \\ \quad \mathit{Kind}(MST_c) = \texttt{concrete} \\ \neg \mathit{MayBeAmbig}(MST', MST_m) \end{array}$$

**Fig. 12.** Type assignments

The metarule (44) defines the type of an interface, which is a pair consisting of an empty fields type and a methods type. This type consists of the sum of the methods types of the superinterfaces, updated by the methods declared in the interface.

The metarule (45-Jam) defines the type of a mixin, which is a pair consisting of a fields and a methods type. The fields type consists of the inherited fields type, updated by the defined fields type; the methods type consists of the sum of the methods types of the implemented interfaces, updated by the inherited methods, updated in turn by the defined methods.

Fig.13 shows the definition of the predicate *MayBeAmbig*, which is true whenever the two arguments type may cause ambiguity (hence make incompatible two methods in mixin instantiation as explained in Sect.2.3). The function *Args* denotes the obvious projection for method signatures.

$$IsPrim(T) =_{\mathrm{def}} T \in \{\texttt{boolean}, \texttt{int}\}$$
$$IsRef(T) =_{\mathrm{def}} \neg IsPrim(T)$$

$$MayBeAmbig(AT_1, AT_2) =_{\mathrm{def}} \begin{cases} AT_1 = T'_1 \dots T'_n \\ AT_2 = T''_1 \dots T''_n \\ \exists i : T'_i \neq T''_i \\ \forall i \in \{1, \dots n\}\ IsRef(T'_i) \Rightarrow IsRef(T''_i) \wedge \\ \qquad\qquad IsPrim(T'_i) \Rightarrow T'_i = T''_i \end{cases}$$

$$MayBeAmbig(MST, MST') =_{\mathrm{def}} \begin{cases} MST = \{msig_1 : MT_1, \dots, msig_n : MT_n\} \\ MST' = \{msig'_1 : MT'_1, \dots, msig'_k : MT'_k\} \\ \exists i \in \{1, \dots, n\}, j \in \{1, \dots, k\}\ t.c. \\ msig_i = msig'_j \wedge \\ MayBeAmbig(Args(msig_i), Args(msig'_j)) \end{cases}$$

**Fig. 13.** Definitions of *MayBeAmbig*

The three metarules (46-48) define the type of a class, which is a pair consisting of a fields and a methods type.

46. For simplicity, we have ignored all the predefined methods of `Object`, defined in [15] 20.1.
47. For a standard Java heir class, the fields type consists of the fields type of the superclass updated by the fields declared in the class. The methods type consists of the sum of the methods types of the implemented interfaces, updated by the methods of the superclass, updated in turn by the methods declared in the class.
48. For a mixin instance, the fields type consists of the fields type of the superclass updated by the fields defined in the mixin. The methods type consists of the sum of the methods types of the implemented interfaces, updated by the methods of the superclass, updated in turn by the methods defined in the mixin.

We assume that the metarules in Fig. 12 can be instantiated only in the cases where update operations are defined.

*Well-formedness of environments* The metarules in Fig. 14 express the fact that a Jam environment is well-formed. More precisely, the judgment $\Gamma \vdash \Gamma'\diamond$ denotes that the declarations in $\Gamma'$ are well-formed in the larger environment $\Gamma$. We follow the approach in [11] of considering larger environment in order to correctly deal with mutual recursion between declarations. An environment $\Gamma$ is well-formed if $\Gamma \vdash \Gamma\diamond$; in this case we will also use the abbreviation $\Gamma \vdash \diamond$.

Fig.14 shows the metarules defining the well-formedness of class, interface and mixin declarations.

| | | |
|---|---|---|
| (30) | $\overline{\Gamma \vdash \emptyset \diamond}$ | |
| (31) | $\dfrac{\Gamma \vdash \Gamma' \diamond \;\; \Gamma \vdash C : FST\ MST}{\Gamma \vdash \Gamma' \cup \{C \text{ is}_c\ CT, C <_c^1 C', C \lhd_i^1 I_1, \ldots, C \lhd_i^1 I_n\}\diamond}$ | $\Gamma'(C) = \bot$ <br> $\Gamma \not\vdash C' \leq_c C$ |
| (32) | $\dfrac{\Gamma \vdash \Gamma' \diamond \;\; \Gamma \vdash C \ :\ FST\ MST}{\Gamma \vdash \Gamma' \cup \{C \text{ is}_c\ K\ KST\ \emptyset\ \emptyset, C <_c^1 C', C \lhd_m M, C \lhd_i^1 I_1, \ldots, C \lhd_i^1 I_n\}\diamond}$ | $\Gamma'(C) = \bot$ <br> $\Gamma \not\vdash C' \leq_c C$ |
| (33) | $\dfrac{\Gamma \vdash \Gamma' \diamond \;\; \Gamma \vdash I : FST\ MST}{\Gamma \vdash \Gamma' \cup \{I \text{ is}_i\ IT, I <_i^1 I_1, \ldots, I <_i^1 I_n\}\diamond}$ | $\Gamma'(I) = \bot$ <br> $\forall j \in \{1, \ldots, n\}\ \Gamma \not\vdash I_j \leq_i I$ |
| (34-Jam) | $\dfrac{\Gamma \vdash \Gamma' \diamond \;\; \Gamma \vdash M : FST\ MST}{\Gamma \vdash \Gamma' \cup \{M \text{ is}_m\ MXT, M \lhd_i^1 I_1, \ldots, M \lhd_i^1 I_n\}\diamond}$ | $\Gamma'(M) = \bot$ |

**Fig. 14.** Well-formed class, interface and mixin declarations

## 4 Jam to Java translation

In this section, we give a formal definition of the dynamic semantics of Jam directly by translation in Java. The same approach of defining a Java extension by translation into Java as been taken for Pizza [18], a superset of Java which incorporates parametric polymorphism, higher-order functions and algebraic data types, and its recent evolution GJ (for "Generic Java") [10].

We first illustrate informally the basic ideas through some examples (Sect.4.1), then provide the formal definition (Sect.4.2); finally in Sect.4.3 we state that the translation preserves static correctness.

### 4.1 An informal overview

The translation from Jam to Java must be defined in such a way to correspond to the informal Jam semantics we have illustrated in Sect.2. Hence, the two basic properties of mixins must be preserved, that is:

- the behavior of a class `H` obtained by instantiating a mixin `M` on a parent `P` must be "equivalent" to that of a class obtained extending `P` by all the defined components of the mixins (copy principle);
- mixin names can be used as reference types (independently from the existence of some mixin instance), and every class which is instance of a mixin must be subtype of both the mixin and the parent type.

The first point immediately gives an easy translation directive, that is, every instantiation of a mixin `M` on a parent `P` must be expanded to a usual Java declaration of a class extending `P` and declaring all the defined components of `M` (plus the constructors possibly declared in the instantiation).

The second point is less trivial to be achieved. Indeed, mixin types in Jam are a new kind of types, not existing in Java, hence they must be translated in either class or interface types.

A simple way to get "for free" the property that a mixin instance turns out to be a subtype of both the mixin and the parent type is to translate a mixin declaration by an interface declaration, and every instantiation by a Java class which (besides extending the parent) implements this interface; however, this choice introduces the problem that mixins in

Jam can declare fields, while interfaces cannot (static components do not cause an analogous problem since they are not part of the mixin type, see Sect.2.2, but only need to be copied at every instantiation).

On the other side, translating a mixin declaration by a class declaration would have the advantage to make possible the declarations of fields, but would require to simulate in Java the implicit Jam type conversion from the mixin instance type to the mixin type.

Hence, we have adopted the first choice, solving the problem of field declarations by a quite standard technique, which is the simulation of fields by a pair of *accessor* methods, for selecting (*getter*) and updating (*setter*) a field. For each field `f` in a mixin declaration, the methods `_$get$_f` and `_$set$_f` are declared in the interface corresponding to the mixin declaration; then, in every class translating a mixin instance, `f` is declared as a field and the two methods are implemented in the obvious way.

Let us now illustrate how the translation works in practice on the mixin `Undo` introduced in Sect.2.1.

```
interface Parent$Undo {
 String getText() ;
 void setText(String s) ;
}
interface Undo extends Parent$Undo {
 // Field "lastText"
 String _$get$_lastText() ;
 String _$set$_lastText(String newValue) ;
 // Methods:
 void setText(String s) ;
 void undo() ;
}
```

**Fig. 15.** Translation of a mixin declaration

Note that, in the translation (shown in Fig.15), together with the interface `Undo` corresponding to the mixin type, there is a second interface `Parent$Undo` which is extended by `Undo` and contains only the declarations of inherited methods.

This interface represents the translation of the type *Parent*(`Undo`) introduced in Sect.2.4, that is, the generic parent type on which the mixin can be instantiated, and is necessary for the Java translation to correctly simulate the Jam extended rule for overloading resolution (an inherited method in a mixin `M` must be considered as it had been declared in a "generic" superclass of `M`, hence considered less specific of a defined method with the same signature).

We consider now an instantiation of the mixin `Undo` in Fig.16, and the corresponding translation in Fig.17.

As shown by the example, the class translating an instantiation of the mixin `M` on a parent `P` extends `P` and implements the interface `M`; moreover, the class contains a copy of all the fields and methods defined in `M`, including static members and abstract methods, and the implementation of the accessor methods for each field.

Note that, inside the mixin, no accessor invocation is used in the translation for accessing the fields. On the contrary, this invocation is necessary in case of access from external code. Moreover, using accessors is only needed when the field access is on an expression of the mixin type, while the code remains unchanged if the expression type is a mixin instance type. For instance, the following code would be kept as it stands by the translation process:

```
ExampleWithUndo e = new ExampleWithUndo() ;
System.out.println(e. lastText) ;
```

```
class Example {
 String donald = "duck" ;
 String getText() {
  return donald ;
 }
 void setText(String donald) {
  this.donald = donald ;
 }
}

class ExampleWithUndo = Undo extends Example {}
```

**Fig. 16.** Undo instantiation example

```
class Example {
 // ... (unmodified)
}

class ExampleWithUndo extends Example implements Undo {
 // Field "lastText"
 String lastText ;
 String _$get$_lastText() {
  return lastText ;
 }
 String _$set$_lastText(String newValue) {
  return lastText = newValue ;
 }
 //
 void setText(String s) {
  lastText=getText() ;
  super.setText(s) ;
 }
 void undo() {
  setText(lastText) ;
 }
}
```

**Fig. 17.** Translation of a mixin instantiation

*Inherited fields* Although inherited fields logically differ from defined fields, they are translated in exactly the same way: a pair of method accessors is generated.

*Static fields* As shown in Sect.2.2, static fields do not belong to the mixin type. Therefore declarations of static fields within a mixin matter only for mixin instances. As a consequence, the pair of interfaces corresponding to a mixin does not contain any accessor for static fields. Instead, static fields will be inserted in every class corresponding to a mixin instance.

### 4.2 Formal translation

In this section we formally define the translation of Jam into Java outlined above. The aim is twofold. First, we define in this way the dynamic semantics of Jam. Second, we get the soundness of the Jam type system from the soundness of the Java type system [11] and Theorem 1 which states that the translation preserves the static semantics.

As usual, for proving preservation of static correctness we need to provide a formal translation not only for Jam programs (environments and body declarations), but also for all judgments, hence for type assertions. In this paper, for lack of space, we omit translation clauses related to body declarations.

*Translation of environments* We denote by $[\![ \Gamma ]\!]$ the translation of a Jam environment $\Gamma$.

Since assertions in $\Gamma$ may be mutually recursive, analogously to what happens for the static semantics, the translation of $\Gamma$ (Fig. 18) uses an auxiliary function taking an additional argument which is a larger environment.

$$
\begin{aligned}
&[\![ \Gamma ]\!] = [\![ \Gamma ]\!]_{\Gamma} \\
&[\![ \gamma_1, \ldots, \gamma_n ]\!]_{\Gamma} = \bigcup_{i \in \{1,\ldots,n\}} [\![ \gamma_i ]\!]_{\Gamma}
\end{aligned}
$$

**Fig. 18.** Translation of environments

The translation of a Jam type assertion is a set of Java type assertions, and is defined in Fig.19. For all the type assertions $\gamma$ for which there is no translation clause, we implicitly assume that:
$[\![ \gamma ]\!]_{\Gamma} = \{\gamma\}$

$$
\begin{aligned}
&[\![ T \lhd_i^1 I ]\!]_{\Gamma} = \{T <_i^1 I\} \text{ if } T \in \mathit{Mixins}(\Gamma) \\
&[\![ T \lhd_i I ]\!]_{\Gamma} = \{T \leq_i I\} \text{ if } T \in \mathit{Mixins}(\Gamma) \\
&[\![ C \lhd_m M ]\!]_{\Gamma} = \{C \lhd_i^1 M\} \\
&[\![ M \text{ is}_m\ FST\ MST\ \texttt{inherited}\ FST'\ MST' ]\!]_{\Gamma} = \\
&\quad \{M \text{ is}_i\ \mathit{AccessorDecs}(FST'[FST]) \cup MST, \mathit{Parent}(M) \text{ is}_i\ MST', M <_i^1 \mathit{Parent}(M)\} \\
&[\![ C \text{ is}_c\ K\ KST\ \emptyset\ \emptyset ]\!]_{\Gamma} = \{C \text{ is}_c\ K\ KST\ FST MST\} \\
&\quad \text{if } \Gamma \vdash C\ \lhd_m\ M, \Gamma \vdash M \text{ is}_m\ FST\ MST\ \texttt{inherited}\ FST'\ MST' \\
&[\![ C\ :\ FST_c\ MST_c ]\!]_{\Gamma} = \{C : FST_c\ MST_c \cup \mathit{AccessorDecs}(FST'[FST])\} \\
&\quad \text{if } \Gamma \vdash C\ \lhd_m\ M, \Gamma \vdash M \text{ is}_m\ FST\ MST\ \texttt{inherited}\ FST'\ MST' \\
&[\![ M\ :\ FST_m\ MST_m ]\!]_{\Gamma} = \{M :\ \emptyset\ MST_m \cup \mathit{AccessorDecs}(FST'[FST])\} \\
&\quad \text{if } \Gamma \vdash M \text{ is}_m\ FST\ MST\ \texttt{inherited}\ FST'\ MST'
\end{aligned}
$$

**Fig. 19.** Translation of type assertions

The translation of the type assertions having form $T \lhd_i^1 I$ and $T \lhd_i I$ depends on the type of the module $T$. If $T$ is a mixin then the assertions are translated into subinterface assertions, otherwise, if $T$ is a class, they remain the same. The instantiation assertion becomes an implementation assertion. A mixin declaration is transformed into the declaration of two interfaces (the first being a subinterface of the second). A class declaration $C$ is modified only in the case $C$ is an instance of a mixin $M$; the translation in this case corresponds to the copy principle. A well-formed mixin type is translated into the two corresponding well-formed interface types. Finally, type assignments for mixin instances and mixins are translated by introducing accessors. Of course, we assume that there are no name conflicts between accessors and user defined methods. The function *AccessorDecs* is defined in Fig.20.

$$
\begin{array}{l}
\mathit{AccessorDecs}(f_1 : FT_1, \ldots, f_n : FT_n) = \bigcup_{i \in \{1,\ldots,n\}} \mathit{AccessorDec}(f_i : FT_i) \\
\mathit{AccessorDec}(f : \texttt{static}\ ST) = \emptyset \\
\mathit{AccessorDec}(f : \texttt{instance}\ ST) = \{\texttt{\_\$get\$\_}f, \epsilon{:}\texttt{instance}\ ST\ \texttt{throws}\ \emptyset, \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{\_\$set\$\_}f, ST{:}\texttt{instance}\ ST\ \texttt{throws}\ \emptyset\ \}
\end{array}
$$

**Fig. 20.** Definition of *AccessorDecs*

### 4.3 Soundness of the Translation

In this section our aim is to show that the translation from Jam into Java is actually a "good" translation.

First of all, it is immediate to see that the translation is conservative, in the sense that every Java program is translated into itself.

More importantly, we state that the translation preserves the static semantics, in the sense that a statically correct Jam program is translated into a statically correct Java program. In order to prove that, we need the stronger property that every valid Jam judgment is translated into a set of valid Java judgments.

**Theorem 1.** *Let $\Gamma$ be a well-formed Jam environment (that is, $\vdash \Gamma\diamond$ is valid), then:*

1. *for every valid judgment $\Gamma \vdash \gamma\diamond$, $[\![\gamma]\!]_\Gamma$ is well-defined and $[\![\Gamma]\!] \vdash [\![\gamma]\!]_\Gamma$ is valid.*
2. *$[\![\Gamma]\!]$ is a well-formed Java environment (that is, $\vdash [\![\Gamma]\!]\diamond$ is valid).*

The proof can be found in [1].

## 5 Conclusion

In the preceding sections, we have described Jam, a smooth extension of Java supporting mixins, and we have formally defined its static semantics and a translation into Java. The latter has been implemented by a Jam to Java translator which makes Jam executable on every platform implementing a Java Virtual Machine.

In this last section, we briefly describe the implementation (Sect.5.1), provide some detailed comparison with related work (Sect.5.2) and discuss some alternative design choices and directions for further investigation (Sect.5.3).

### 5.1 Implementation

The translator (called `jamc`) has been implemented in Java.

It performs a complete syntactic analysis and only a partial type-checking of Jam input source files. This means that every lexical or syntactic error in the source code will be detected by `jamc`, whereas the most of static errors will be found later on by the Java compiler when trying to compile the Java source files produced by `jamc`.

For more details see [1].

### 5.2 Related work

At our knowledge, the only existing proposals for extensions of object-oriented languages with mixins are [9] and [14].

In [9], the authors present an extension of Smalltalk with mixins. The design principles of this extension are very similar to those we have followed in Jam. Indeed, mixins are seen as functions from superclasses into heir classes, instantiation is possible only if the candidate parent class contains all the methods invoked via `super` in the mixin, mixins do not influence the behavior of existing Smalltalk programs, hence the extension is fully upward-compatible. The great difference is that, being Smalltalk an untyped language, most of the problems we had to face in the design of Jam simply do not exist for Smalltalk; the most remarkable of these problems is that mixins introduce a new kind of reference type. As in our approach (see Sect. 2.3) overriding takes place uniformly both for methods which are invoked via `super` and for others. Following our same principle that mixin instantiation should produce a correct heir class, the candidate parent class must not contain instance variables with the same name of some defined in the mixin (indeed in Smalltalk hiding parent variables is forbidden). Moreover, mixins can be easily eliminated from a program by automatically creating a class for each mixin invocation and duplicating the mixins code for it (in other words, mixins have a pure copy semantics, corresponding to $\beta$-rule for function application), while for Jam this is not enough since mixins are *types* so they cannot be just eliminated.

In [9], a mixin can be composed with another mixin (the expected semantics is exactly function composition) and a mixin can also be "extracted" from an existing class: in this case, its components are those declared in the class. Both the possibilities seem very useful and adding them to Jam will be matter of further work, even though a generalization allowing full mixin composition seems in the Java case not trivial, on both design and implementation side.

The authors have developed a working extension which has been used for real applications.

In [14], the authors describe MixedJava, a theoretical language which has a Java-like syntax where it is only possible to declare either mixins or interfaces, while usual classes are seen as particular mixins which define all the components.

In MixedJava, there are two kinds of mixins.

- *Atomic* mixins, whose declaration, similar to that of a usual Java class, contains fields, methods and an interface which specifies the expected superclass. This interface plays the same role of the `inherited` part of mixins in Jam, with the difference that it must be explicitly declared by the programmer, while in Jam the interface is created during the translation process.
  A basic difference (see Sect.2.3) is that in mixin instantiation (which in MixedJava is just a special case of mixin composition, see below) methods in the heir override methods in the parent *only if* they are explicitly mentioned in the inheritance interface, while in case of unexpected overriding both the versions are kept.
- *Compound* mixins, roughly based on function composition, as happens for the Smalltalk extension described above, but actually more involved, for the constraints on method overriding explained above.

The work presented in [14] sensibly differs from ours for many reasons.

- The proposed language is theoretical, while Jam is designed to be a working upward-compatible extension of Java (1.0).
- In MixedJava inherited components can be only methods, since they are specified via an interface. The authors motivate this choice by the consideration that programming via interfaces is cleaner; in Jam, we have chosen as privileged principle that mixins should be similar as much as possible to usual heir classes.
- In Jam mixins can be only instantiated on classes, and there is no notion of mixin composition. As already stated, this is an important possibility of extension of Jam to be investigated in the future.

– As mentioned above, MixedJava adopts an ad-hoc solution in the case of unexpected overriding, while in Jam methods in the parent class are uniformly overridden by methods in the heir class. This different policy is probably the most important difference between the two approaches. A disadvantage of our approach is that in the case the parent class incidentally has some method which is in conflict with one defined in the mixin, it is left to the user the choice between either to avoid this instantiation (hence the mixin becomes useless for this particular case) or to get an heir class with some overriding which was not planned when designing the mixin. However, the conflicts resolution in [14], essentially based on the idea of keeping both the method versions, leads as a matter of fact to ambiguity problems which are typical of multiple inheritance (a class inherits two different definitions for the same method), heavily complicating both language semantics and a possible implementation (only outlined in [14]). On the contrary, our choice implies minimal changes w.r.t. Java semantics. A future development could be the analysis of intermediate solutions.

### 5.3 Alternative design choices and further developments

**References to the parent and the heir names** In Sect. 2.6 we have seen that there are cases where heir classes cannot be "abstracted" in a mixin definition. Introducing canonical notation for the parametric names of the parent and heir class, say `P*` and `H*`, respectively, we could transform the class `H` shown in Sect. 2.6 in a mixin `M` as follows.

```
mixin M {
 inherited static int counter ;
 static int counter ;
 static void incrThat() { ++P*.counter ; }
 ...
 int value ;
 public boolean equals(Object that) {
  if (that instanceof H*) return ((H*)that).value == value ;
  return false ;
 }
}
```

Obviously, the copy principle should in this case be modified, saying that a class `H = M extends P` should be equivalent to a class extending `P` and containing the definitions in `M` where all the occurrences of the parametric names `P*` and `H*` have been replaced by `P` and `H`, respectively. Introducing this possibility would allow a (limited to heir classes) form of parametric polymorphism, in the same direction of the extensions of Java with parametric types proposed in [18,10]. However, with this choice we would lose one of the two design principles of Jam, that is, the fact that a mixin name can be used as a type (indeed in this case it would be not a type but a *type schema*), hence all the mixin instances can be uniformly used through the common interface specified by this type. Indeed, it is not clear if it could be possible (and how) to make compatible these two different ways of achieving abstraction: on one side to have parametric modules (class-to-class functions) where this parametricity is fully exploited, on the other side to be able to use each module as a type. The problem is not trivial and deserves further investigation.

**Flexible matching** Assume that `P` is a supertype of `H` and consider the following declarations.

```
mixin M {                          class C1 {
 inherited void f(H, H) ;           void f(P p, H h) {}
}                                  }
```

In Jam it is not possible to instantiate M on C1 since this class does not provide an implementation for the method `void f(H,H)`. Indeed, the matching between the `inherited` methods and the corresponding methods in the parent class is required to be *exact* (same arguments and return type, and equivalent `throws` clause). An interesting possibility, which could be matter of a future extension, could be to introduce a *flexible* matching, where the subtyping rule for method types (Fig. 11, metarule (22)) allows contravariance on arguments type and covariance on return type. On the contrary, it is interesting to note that the exception types must be invariant (modulo the equivalence $=_e$) in order to preserve the soundness of the type-system.

Allowing this flexibility, C1 turns out to be a correct parent class for M. However, this kind of matching leads to some new problems w.r.t. the exact matching case. Let us consider this other class declaration.

```
class C2 {
 void f(P p, H h) {}
 void f(H h, P p) {}
}
```

In this case, assuming that we want to instantiate M on C2, we have to decide in some way which of the two methods declared in C2 must be used as implementation of the `inherited` method in M. The choice could either be driven, in analogy with the overloading resolution in Java, by the notion of most specific applicable method, or left to the user via a mechanism which permits to explicitly specify in the instantiation the association of `inherited` methods with those defined in the parent class.

**Shared static components** In Sect.2.2, we have seen that each class has its own copy of the static components declared in the mixin. As already mentioned there, other two design choices would be possible: either make mixin instances to share a unique copy for each static component (in this way they would be part of the mixin type), or leave to the user, by means of a keyword `shared` or analogous mechanism, the choice between the two options. This last choice, which has some appeal, would require the introduction of some constraint, for instance the fact that a `shared static` method could not invoke a `static` method.

## References


1. D. Ancona, G. Lagorio, and E. Zucca. Jam - a smooth extension of java with mixins. Technical report, DISI, University of Genova, 1999. Available at `http://www.disi.unige.it/ftp/person/AnconaD/Jam.ps.gz`.
2. D. Ancona and E. Zucca. A theory of mixin modules: basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, 1998.
3. D. Ancona and E. Zucca. A primitive calculus for module systems. In G. Nadathur, editor, *Principles and Practice of Declarative Programming, 1999*, Lecture Notes in Computer Science, pages 62–79. Springer Verlag, 1999.
4. G. Banavar and G. Lindstrom. An application framework for module composition tools. In *ECOOP '96*, number 1098 in Lecture Notes in Computer Science, pages 91–113. Springer Verlag, July 1996.
5. Grady Booch. *Object-Oriented Analysis and Design.* Addison-Wesley, 1994.
6. G. Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance.* PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.
7. G. Bracha and W. Cook. Mixin-based inheritance. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1990*, pages 303–311. ACM Press, October 1990. SIGPLAN Notices, volume 25, number 10.
8. G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proc. International Conference on Computer Languages*, pages 282–290, San Francisco, April 1992. IEEE Computer Society.
9. G. Bracha and G. Lindstrom. Extending Smalltalk with mixins. In *OOPSLA96 Workshop on Extending the Smalltalk Language*, April 1996. Electronic note available at `http://www.javasoft.com/people/gbracha/mwp.html`.

10. G. Bracha, M. Odersky, D. Stoutmire, and P. 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*, October 1998. Home page: `http://www.cs.bell-labs.com/who/wadler/pizza/gj/`.
11. S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lecture Notes in Computer Science, pages 41–82. Springer Verlag, Berlin, 1999.
12. S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java type soundness revisited. Technical report, Dept. of Computing - Imperial College of Science, Technology and Medicine, October 1999.
13. D. Duggan and C. Sourelis. Mixin modules. In *Intl. Conf. on Functional Programming*, pages 262–273, Philadelphia, June 1996. ACM Press. SIGPLAN Notices, volume 31, number 6.
14. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *ACM Symp. on Principles of Programming Languages 1998*, pages 171–183, January 1998.
15. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.
16. S.C. Keene. *Object Oriented Programming in Common Lisp: A Programming Guide in CLOS*. Addison-Wesley, 1989.
17. D.A. Moon. Object oriented programming with Flavors. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1986*, pages 1–8, 1986.
18. M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *ACM Symp. on Principles of Programming Languages 1997*. January 1997.
19. A. Snyder. CommonObjects: An overview. *SIGPLAN Notices*, 21(10):19–28, 1986.