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)
{davide,lagorio,zucca}@disi.unige.it

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, except 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 somehow similar to that based on multiple inheritance, but avoids the associated complications.
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 [17,15]) has attracted 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 *`decs`*.

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

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 154–178, 2000.

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

Consider now a language allowing to give a name, say `M`, to ***decs***, and instantiating `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 ;
```

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

A mixin declaration resembles a usual heir class declaration, except for the fact that a mixin does not refer to a fixed parent class, but simply specifies the set of fields and methods a 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 [5]. This paradigm is similar to multiple inheritance, but avoids the associated complications.

Mixin-based programming has been now extensively studied both on the methodological and foundational point of view [6,5,2,3]. 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. Then, 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 tried to 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 [15,19]. After that, at our knowledge, there exist only a proposal for extending ML [12], a working extension of Smalltalk [7] and a proposal for a Java-like mixin language [13] (whose relation with our work will be discussed in detail in Sect.6).

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

[1] **Ja**va + **m**ixin = **Jam**

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 an introduction to Jam, through some examples, and illustrate and motivate in detail our design choices. In Sect.3 we outline the abstract syntax and the formal static semantics (the full definition can be found in [1]). In Sect.4 we define a 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). In Sect.5 we briefly describe the implementation. Finally in Sect.6 we provide a detailed comparison with related work 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 its sources are available at the following URL: `http://www.disi.unige.it/person/LagorioG/jam`.

2 Introduction and Rationale

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 supports restoring of the text before the latest modification.

```
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

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 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

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

Fig. 2. Mixin instantiation

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 (substitutable)[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.

This principle corresponds to the "natural" semantics for mixin instantiation, that is, the semantics one would obtain defining a "hand-made" subclass. 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 demonstrates 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 is that Jam supports a programming style (sometimes called mixin-based [6]) where different ingredients are "mixed" together in defining a class. This paradigm has been advocated [5] on the methodological

[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 by instantiating a mixin, to be not confused with an *instance* of a class.

```
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

side since it allows to recover some of 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 many interfaces in exactly the same way a class does.

Constructors. In Jam constructors cannote be declared in a mixin, but only for each single mixin instance at the point of instantiation. From a technical point of view, it would be conceivable to declare constructors in mixins, handling them as components which are not part of the mixin type, as we do for static components (see below). However, we have preferred this choice since from a methodological point of view constructors are tightly tied-up with the implementation of their own class, so their signatures tend to be not very general.

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()` with *e* of type `H` are legal. The same rule holds for fields[4]. 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 accessing static members through the mixin type should be allowed as well;
- 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.

2.3 Constraints on Instantiation

As mentioned in Sect. 1, the fact that a class `P` provides an implementation for the `inherited` part of a mixin `M` is not enough to ensure 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.

Illegal overriding/hiding A method in `P` is illegally overridden (hidden) (see 8.4.6.3 in [14]) by a defined method in `M` if it has the same signature (name and arguments type) but either different return type, or different kind (`instance` or `static`) or incompatible `throws` clause. This is not correct in Jam as well as in Java.

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 and a compatible (see 8.4.4 in [14]) `throws` clause. For instance, instantiating the mixin `Undo` on a class with a `void undo()` method produces an unexpected overriding. This situation looks somehow undesirable, since there is some overriding which was not planned when declaring the mixin; however, our choice for Jam has been to consider these instantiations to be legal, leaving to the programmer the responsibility of avoiding them when the additional overriding is undesired[5]. Indeed, different choices would sensibly complicate either

[4] We maintain this alternative syntax for compatibility reasons only, see 15.10.1 of [14].

[5] A Jam compiler may issue a warning in such cases.

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 usual heir classes semantics. See Sect.6 for some further discussion on this point.

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)`, also another 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 (see 15.11.2 in [14]).

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. Alternatively 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).

2.4 Overloading

The Java rules for overloading resolution (see 15.11.2 in [14]) smoothly extend 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 special care is needed for handling the situation when there is an overloading conflict between an inherited and a defined method in a mixin. We illustrate the problem in terms of the following example.

```
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 another 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 (see 15.11.2.2 in [14]).

If we now consider 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.

In order to achieve this goal, we assume that inherited methods in a mixin `M` are annotated with a type (that is, considered to be 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 it 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 (compile-time) type `C` in Java (see 15.7.2. in [14]). 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 its (possibly 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, 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`.

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: in a mixin declaration, in Jam, there is no way to refer to the parametric types of either the parent or

```
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

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 invocations 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 class `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` (this could be achieved by extending the use of the keyword `super` to such cases).
- The method `equals` uses the name of the heir class `H` which is unknown for a mixin (note that this cannot be achieved using `M`, since that would be kept as `M` in all mixin instances following the copy principle).

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.6 we discuss possible solutions to this problem.

3 Outline of The Formal Definition

In this section we outline the abstract syntax and the static semantics of Jam; the full definition can be found in [1].

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); an extension to Java 1.1 would require some more work at the implementative level, but seems in principle not to introduce new problems. Indeed the main enhancement in Java 1.1 is the introduction of inner classes, whose semantics is specified by flattening them to usual top-level classes.

The formal definition in [1] only considers a subset of the language chosen to be a minimal but sufficient set 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 stated in the next section). Excluded features fall under two main categories: those which are orthogonal with our aim, like multithreading and inner classes 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 [10]: constructors, `static` members, checked exceptions[6], `abstract` classes and methods, method invocations and field accesses via `super`.

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.

Fig. 6 contains the part of the Jam abstract syntax related to interface, class and mixin declarations (the Jam specific productions are the first three only).

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

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

Fig. 6. Jam abstract syntax

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.

In Fig. 7 we define the Jam types. A generic *type* can be a reference type, a primitive 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) 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 types. The *constructors type* is a non-empty set of constructor types. Note that every class has always at least one constructor (if it is not explicitly given the default one is assumed).

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

Fig. 7. Jam types

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.

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 [11], we consider two components that can be extracted in a trivial way from a program: the environment Γ, 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 is contained in Γ). The syntax of these two components is given in Fig. 8.

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

Fig. 8. Environments and body declarations

We assume that in the environment extraction process a check is performed for avoiding duplicate declarations. Hence, the static correctness of a Jam program can be expressed by the validity of the two following judgments:

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

The former means that Γ 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 Γ. The validity of these two judgments is defined inductively introducing other judgments relative to subcomponents. Because of lack of space, here we omit all the typing rules definining the validity of the second judgment and only show some significant typing rules defining the validity of the first judgment (well-formedness of environments), in particular all those Jam-specific; for the missing metarules we provide a brief informal explanation. The full type system can be found in [1].

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

- C is$_c$ K KST FST MST : the class C of kind K declares the specified constructors (KST), fields (FST) and methods (MST)
- $C <_c^1 C'$: the class C directly extends the class C'
- $T \lhd_i^1 I$: 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
- $I <_i^1 I'$: 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

Judgments of the metarules related to well-formedness of environments have the form $\Gamma \vdash \gamma$, with Γ an environment and γ a *type assertion*.

A group of metarules of the Jam type system defines 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 $<_c^1$ is the *subclass* relation $\leq_c$; analogously the reflexive (on existing interface types) and transitive closure of $<_i^1$ is the *subinterface* relation $\leq_i$. The *implementation* relation from classes to interfaces is derived from $\lhd_i^1$ 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* (subtyping) between reference types, denoted by $\leq$. In Fig. 9 we show the metarule stating that a mixin instance is a subtype of the corresponding mixin type and that expressing reflexivity of widening for mixin types.

Another group of metarules defines subtyping relations for exceptions, fields, methods and module types. These relations basically express that a module type $MODT$ is a subtype of another module type $MODT'$ (written $MODT \leq_{mod} MODT'$) if it has more fields and/or methods; the common fields and methods must have *exactly* the same type, modulo equivalence of exceptions types $=_e$, defined by $\Gamma \vdash ET =_e ET'$ iff $\Gamma \vdash ET \leq_e ET'$ and $\Gamma \vdash ET' \leq_e ET$. Subtyping relation for exceptions types is defined in Fig. 9; note that it is possible that $E_i = E_j$ or $E'_i = E'_j$ holds for some i, j.

$$\frac{\Gamma \vdash C \lhd_m M}{\Gamma \vdash C \leq M} \qquad \frac{\Gamma \vdash M \text{ is}_m MXT}{\Gamma \vdash M \leq M}$$

$$\frac{\Gamma \vdash E_1 \leq_c E_1' \ldots \Gamma \vdash E_n \leq_c E_n' \quad \Gamma \vdash \{E_1, \ldots, E_n\} \diamond_{\text{ExcType}} \quad \Gamma \vdash \{E_1', \ldots, E_n'\} \diamond_{\text{ExcType}}}{\Gamma \vdash \{E_1, \ldots, E_n\} \leq_e \{E_1', \ldots, E_n'\}} \quad n \geq 0$$

Fig. 9. Relations on types

Other metarules define *type assignments*, that is, the fact that some Jam module (either class or interface or mixin) has a given type. In Fig.10 we show the two metarules related to the two Jam-specific cases, that is a mixin and a class which is a mixin instance.

Set $MXT = FST\ MST$ `inherited` $FST'\ MST'$

$$\frac{\begin{array}{l}\Gamma \vdash M \text{ is}_m MXT \quad \Gamma \vdash MXT \diamond_{\text{MixinType}} \\ \Gamma \vdash I_1 : \emptyset\ MST_1 \ldots \Gamma \vdash I_n : \emptyset\ MST_n \\ \Gamma \vdash M : FST'[FST] \\ (MST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} MST_n \overset{\Gamma}{\oplus} MST')[MST]_\Gamma\end{array}}{\begin{array}{l}n \geq 0 \\ \{I_1, \ldots, I_n\} = \{I \mid M \lhd_i^1\ I \in \Gamma\}\end{array}}$$

Set:

$FST_c = FST'[FST_d], MST_c = (MST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} MST_n \overset{\Gamma}{\oplus} MST')[MST_d]_\Gamma$

$MST_m = (MST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} MST_n \overset{\Gamma}{\oplus} MST_i)[MST_d]_\Gamma, CT = K\ KST\ \emptyset\ \emptyset$

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

Fig. 10. Type assignments

The type of a class, interface or mixin is a module type, that is, a pair consisting of a fields type and a methods type. The first metarule in Fig. 10 defines the type of a mixin. The fields type consists of the inherited fields type, updated by the defined fields. The methods type consists of the sum of the methods types of the implemented interfaces and of the inherited methods, updated by the defined methods. The auxiliary *update* operations on (legal) Jam fields and methods types, written $\_[\_]$, return a new fields (resp. methods) type obtained updating the first argument with new fields (resp. methods), if this is possible, accordingly with Java rules on hiding, overloading and overriding. For instance, updating a methods type is undefined if we try to override a method with another which has the same signature and different return 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 two methods with the same signature the operation is defined only if they have the same return type and either they are both `abstract` or one is `abstract` and the other is an instance method with compatible `throws` clause. In this case, the sum contains just one such method whose exceptions type is the "intersection" of the exceptions types, defined by

$E \in ET \overset{\Gamma}{\otimes} ET'$ iff either $E \in ET$ and $\exists E' \in ET'$ *s.t.* $\Gamma \vdash E \leq_c E'$ or conversely.

This operation is needed in the case a class inherits (from the parent class and implemented interfaces) many methods which differ only for the `throws` clauses.

For the formal definition of update and sum operations see [1].

The second metarule in Fig. 10 defines the type of 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 and the methods of the superclass, updated by the methods defined in the mixin. The predicate *MayBeAmbig* (omitted here) holds whenever the two arguments types may cause ambiguity, and thus make two methods incompatible in mixin instantiation as explained in Sect.2.3.

Other analogous metarules define the type of an interface and of a standard Java class. We assume that all these metarules can be instantiated only when update operations are defined.

In Fig. 11 we show some of the metarules which define when a Jam environment is well-formed. More precisely, the judgment $\Gamma \vdash \Gamma'\diamond$ denotes that the declarations in Γ' are well-formed in the larger environment Γ. Indeed, we follow the approach in [10] of considering larger environment in order to correctly deal with mutual recursion between declarations. An environment Γ is well-formed if $\Gamma \vdash \Gamma\diamond$; in this case we also use the abbreviation $\Gamma \vdash \diamond$.

We show, in particular, the metarules which define well-formedness of mixin declarations and mixin instantiations; analogous rules hold for interface and usual class declarations.

$$\frac{\Gamma \vdash \Gamma' \diamond \quad \Gamma \vdash M : FST\ MST}{\Gamma \vdash \Gamma' \cup \{M \text{ is}_m\ MXT, M \lhd_i^1 I_1, \ldots, M \lhd_i^1 I_n\}\diamond} \quad \Gamma'(M) = \bot$$

$$\frac{\Gamma \vdash \Gamma' \diamond \quad \Gamma \vdash C\ :\ FST\ MST}{\Gamma \vdash\ \Gamma' \cup \{C \text{ is}_c\ K\ KST\ \emptyset\ \emptyset, C <_c^1 C', C \lhd_m M,\ C \lhd_i^1 I_1, \ldots, C \lhd_i^1 I_n\}\diamond} \quad \begin{array}{l}\Gamma'(C) = \bot \\ \Gamma \nvdash C' \leq_c C\end{array}$$

Fig. 11. Well-formed declarations

4 Jam to Java Translation

In this section, we give a formal definition of the dynamic semantics of Jam directly by translation into Java. The same approach of defining a Java extension by translation into Java has 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") [9].

We first illustrate informally the basic ideas through some examples, then provide the formal definition; finally we state that the translation preserves static correctness.

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 by extending `P` by all the defined components of `M` (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 class.

The first point immediately gives an easy translation directive: 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. Indeed, mixin types in Jam are a new kind of types, not existing in Java, hence they must be mapped onto either class or interface types.

A simple way to get the property that a mixin instance turns out to be a subtype of both the mixin and the parent type "for free" 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.

On the other hand, translating a mixin declaration by a class declaration would have the advantage of making 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 the standard technique of simulating fields by pairs of *accessor* methods for selecting (*getter*) and updating (*setter*) fields. For each field `f` in a mixin `M` declaration, the methods `M_$get$_f` and `M_$set$_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[7].

[7] The implementation, for handling Java packages, uses a fully-qualified (as defined in 6.7 of [14]) mixin name, rewritten in some way to eliminate the dots, in the accessor name to avoid possible name clashes.

Let us now illustrate how the translation works in practice on the mixin `Undo` introduced in Sect.1.

```
interface Parent$Undo {
 String getText() ;
 void setText(String s) ;
}
interface Undo extends Parent$Undo {
 String Undo_$get$_lastText() ;
 String Undo_$set$_lastText(String newValue) ;
 void setText(String s) ;
 void undo() ;
}
```

Fig. 12. Translation of a mixin declaration

Note that in the translation (shown in Fig.12), 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.13, and the corresponding translation in Fig.14.

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

Fig. 13. `Undo` instantiation example

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.

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

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

Fig. 14. Translation of a mixin instantiation

Note that, inside the mixin, no accessor invocation is used in the translation for accessing the fields. Such invocations are only necessary when accessing 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) ;
```

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.

Formal translation. We formally define now the translation of Jam into Java. The aim is twofold: first, to define the dynamic semantics of Jam; second, to prove soundness of the Jam type system from the soundness of the Java type system [10] and from 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.

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

Since assertions in Γ may be mutually recursive, analogously to what happens for the static semantics, the translation of Γ (Fig. 15) uses an auxiliary function taking an additional argument which is a larger environment.

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

Fig. 15. Translation of environments

The translation of a Jam type assertion is a set of Java type assertions, and is defined in Fig.16. For all the type assertions γ for which there is no translation clause, we implicitly assume that $[\![\, \gamma \,]\!]_{\Gamma} = \{\gamma\}$.

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

Fig. 16. Translation of type assertions

The translation of the type assertions having form $T \lhd^1_i I$ 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 (that is, if T is a class), they remain unaltered. 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. 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 *Mixins* returns all the mixins declared in a given environment; the function *AccessorHeadings* returns the accessors (getter and setter) corresponding to a set of fields; the function *Abstract*, given a set of methods, returns as `abstract` methods all the instance methods.

We state now that the translation preserves the static semantics, in the sense that a statically correct Jam program is translated into a statically correct Java program. This is a implied by the more general property that every valid Jam judgment is translated into a set of valid Java judgments.

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

1. *for every valid judgment $\Gamma \vdash \gamma$, $[\![\, \gamma \,]\!]_{\Gamma}$ is well-defined and $[\![\, \Gamma \,]\!] \vdash [\![\, \gamma \,]\!]_{\Gamma}$ is valid.*

2. $[\![\Gamma]\!]$ *is a well-formed Java environment (that is,* $[\![\Gamma]\!] \vdash \diamond$ *is valid).*

The proof can be found in [1].

5 Implementation

The translator (called `jamc`), implemented in Java, 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 most static errors will be found later by the Java compiler when trying to compile the Java source files produced by `jamc`. Hence obtaining bytecode from a Jam source involves the execution of two distinct tools: the translator and a standard Java compiler. The former requires the execution of a JVM, so this step might be quite expensive even though the translation process is cheap in computational terms. The cost of the latter step is, of course, the usual cost required by a standard Java program.

In the current implementation, we have chosen to follow the copy principle in a straightforward way, because this particular translation guarantees good performances. In other words, following the terminology introduced by Pizza [18], we have developed a *heterogeneous* translation which tends to favour the running time of the generated code penalizing the bytecode size. Alternatively, a *homogeneous* translation could be considered in order to share as much as possible code between the instantiations of a mixin, at the cost of some time overhead due to casts. These translations would be preferred in environments like embedded systems and smart-card applications where the code size is often a main concern.

We finally mention that `jamc` must deal with two features that we have not discussed in this paper: packages and access modifiers. For more details see [1].

6 Related and Further Work

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 provide some detailed comparison with related work and discuss some alternative design choices and directions for further investigations.

Object oriented languages supporting mixins. To our knowledge, the only existing proposals for extensions of object-oriented languages with mixins are [7] and [13].

In [7] is presented 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 most significant difference is due to the fact that Smalltalk is untyped, and so, 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 β-rule for function application), while for Jam this is not enough since mixins are *types* so they cannot be just eliminated.

In [7], 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 [13] is described 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 as 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 [13] differs significantly from ours. Namely:

- 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 the main principle that mixins should be similar to usual heir classes as much as possible.
- 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 to 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 if the parent class incidentally has some method which is in conflict with one defined in the mixin, it is left to the user 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 [13], essentially based on the idea of keeping both method versions, leads 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 [13]). A future development could be the analysis of intermediate solutions.

Mixins vs. parametric types. A great effort has been spent in the last years by the scientific community in proposing extensions of Java with parametric types, see, e.g., Pizza [18], its evolution GJ [9] and PolyJ [16]; these proposals are presently under consideration of the Java Community Process initiative. With respect to Java extensions with parametric types, extensions with mixins go in a different, in a sense orthogonal, direction. The main disadvantage of the mixin-based approach is that there is no mean in a mixin 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 illustrated in Sect.2.6.

Hence, 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, in this case the copy principle should 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 form of parametric polymorphism (limited to heir classes), similar to the extensions of Java with parametric types mentioned above. 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 the approaches based on parametric types, these cannot be directly used as Java types, but are type schemata; hence it is not possible to uniformly use all their instantiations. It is not clear whether (and how) it is possible to reconcile these two different ways of achieving abstraction: parametric modules (class-to-class functions) where this parametricity is fully exploited, and using modules as types. 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 {
 inherited void f(H, H) ;
}
```

```
class C1 {
 void f(P p, H h) {}
}
```

In Jam instantiating `M` on `C1` is illegal, since `C1` 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 a *flexible* matching, allowing contravariance on arguments type and covariance on return type. However, 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 which of the two methods declared in `C2` to use 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[8], 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. Further enhancements could also include a mean of renaming methods to make them match an `inherited` specification.

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

[8] Hence, in this particular example, there would be an error.

there, other two design choices are conceivable: either mixin instances 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. The latter choice, which has some appeal, would require the introduction of some constraint, for instance the fact that a `shared static` method may not invoke a `static` method.

Acknowledgments. We warmly thank Sophia Drossopoulou and the anonymous referees for the careful reading and useful suggestions for improving the paper.

References

1. D. Ancona, G. Lagorio, and E. Zucca. Jam - A smooth extension of Java with mixins. Technical report, DISI-TR-99-15, University of Genova, November 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. G. Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.
6. 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.
7. G. Bracha and D. Griswold. 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`.
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, 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/`.
10. 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.
11. 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.
12. 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.

13. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *ACM Symp. on Principles of Programming Languages 1998*, pages 171–183, January 1998.
14. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.
15. S.C. Keene. *Object Oriented Programming in Common Lisp: A Programming Guide in CLOS*. Addison-Wesley, 1989.
16. A.C. Meyers, J.A. Bank, and B. Liskov. Parameterized types for Java. In *Proc. 24th ACM Symp. on Principles of Programming Languages*. ACM Press, January 1997.
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.