Jam—Designing a Java Extension with Mixins

DAVIDE ANCONA, GIOVANNI LAGORIO, and ELENA ZUCCA
University of Genoa

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 by instantiating the same mixin are considered subtypes of the corresponding type, and 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 somewhat 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, it means that we have developed a Jam-to-Java translator which makes Jam sources executable on every Java Virtual Machine.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*

General Terms: Languages

Additional Key Words and Phrases: Java, language design

1. INTRODUCTION

In the last years, the notion of *parametric heir class* or *mixin* (following the terminology originally introduced in Moon [1986] and Keene [1989]) has been the subject of 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

This work was partially supported by DART—Dynamic Assembly, Reconfiguration and Type-checking; Murst NAPOLI—Network Aware Programming: Oggetti, Linguaggi, Implementazioni; and APPlied SEMantics—Esprit Working Group 26142.

Authors' address: Dipartimento di Informatica e Scienza dell'Informazione, University of Genoa, Via Dodecaneso, 35, 16146 Genoa, Italy; email: {davide,lagorio,zucca}@disi.unige.it.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2003 ACM 0164-0925/03/0900-0641 $5.00

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* }

Consider now a language allowing us to give a name, say M, to *decs*, and to instantiate M on different parent classes, for example, 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 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 Bracha [1992]. This paradigm is somewhat 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 [Bracha and Cook 1990; Bracha 1992; Bracha and Lindstrom 1992; Banavar and Lindstrom 1996; Ancona and Zucca 1998, 2002]. 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 us to have a clean and unifying view of different linguistic mechanisms for composing modules. Moreover, 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 [Ancona and Zucca 1998, 2002].

Despite 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 [Keene 1989; Snyder 1986]. After that, to our knowledge, there exist only a proposal for extending ML [Duggan and Sourelis 1996], a working extension for Smalltalk [Bracha and Griswold 1996], and a proposal for a Java-like

mixin language [Flatt et al. 1998] (the relation of these last two proposals with our work will be discussed in detail in Section 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), where 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 that makes Jam immediately executable on every Java Virtual Machine.

The structure of the presentation is as follows. In Section 2 we provide a user introduction to Jam, through some examples, and illustrate and show in detail the reasons for our design choices. In Section 3 we formally define the language, giving the abstract syntax and the static semantics. 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 Drossopoulou and Eisenbach [1999], even though we cover some more features and take a somewhat different style of presentation. In Section 4 we define a formal translation from Jam into Java and state the correctness of this translation with respect to static semantics (that is, correct Jam programs are expanded into correct Java programs; this also ensures the soundness of the Jam type system). In Section 5 we give a short description of the implementation. In Section 6 we provide a detailed comparison with the proposals in Bracha and Griswold [1996] and Flatt et al. [1998] and with the Java extensions with parametric types in Odersky and Wadler [1997], Bracha et al. [1998], and Meyers et al. [1997]; moreover, we outline further research directions.

The Jam compiler, its sources, and the complete LALR grammar are available online at: `http://www.disi.unige.it/person/LagorioG/jam`.

This paper is an extended version of Ancona et al. [2000].

2. INTRODUCTION AND RATIONALE

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

We will denote references to a specific section in the Java Language Specification [Gosling et al. 2000] by JLS followed by the section number.

[1] Java + mixin = Jam.

```
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.

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

class TextboxWithUndo = Undo extends Textbox {}
```

Fig. 2. Mixin instantiation.

2.1 An Example

Figure 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. 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 happens for usual heir classes.[2]

The mixin `Undo` can be instantiated on classes that define two nonabstract methods `getText` and `setText`, with types as specified in the `inherited` declaration. Figure 2 shows an example of instantiation; we have used as parent a class `Textbox` that extends a generic class `Component` (these classes could be part of a GUI[3] class library). 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.

[2]When the overridden/hidden component is a method, it must explicitly be declared `inherited` (see also Section 2.3.).
[3]**G**raphical **U**ser **I**nterface.

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 type, the same arguments type, and equivalent (substitutable)[4] `throws` clause with respect to 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 by 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 a simple and natural semantics of mixins from the point of view of the programmer: by mixin instantiation, it is possible to get the same effect one should obtain in pure Java by defining many "hand-made" subclasses which extend different parent classes by the same features.

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 by instantiating the mixin must be a correct Java heir class. This leads to some restrictions described in detail in Section 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 it is possible for the programmer to avoid writing duplicated code, a good result in itself, but Jam allows something more: a mixin can be used as a *type* and a mixin instance[5] is a subtype of both the mixin and the parent class on which it has been instantiated. More precisely, the language provides the following features:

(1) Mixin names (`Undo` in the example) can be used as (reference) types; therefore, besides class and interface types, in Jam there exists a new kind of reference types called *mixin types*.
(2) A mixin type can be used independently from the existence of any mixin instance.
(3) Similarly to what happens with other reference types, it is possible to access fields (`lastText` in the example) and invoke inherited (`getText`), defined (`undo`), and redefined (`setText`) methods on expressions of mixin types.

---

[4]That is, every exception declared in one clause must be a subtype of an exception declared in the other, and conversely; this will be formalized in Section 3.5 by an equivalence relation $=_e$ on exceptions types.

[5]We will call *mixin instance* a class obtained by instantiating a mixin, not to be 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.

(4) Every expression whose type is a mixin instance is implicitly convertible[6] to the corresponding mixin type.
(5) Every expression whose type is a mixin instance is implicitly convertible to the parent type on which the mixin has been instantiated.

In particular, point 4 allows the programmer to manipulate objects of any mixin instance by using the common interface specified by the mixin declaration (see Figure 3). On the other hand, the last point is the minimum requirement imposed by the equation *mixin* = *parametric heir class*, which was our starting design principle in developing Jam.

Figure 4 shows graphically the example type hierarchy using UML notation [Booch et al. 1998].

An important consequence of the features illustrated until now is that Jam supports a programming style (sometimes called *mixin-based* [Bracha and Cook 1990]) where different ingredients are "mixed" together in defining a class. This paradigm has been advocated [Bracha 1992] on the methodological side since it allows one 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.

[6]Note that "implicitly convertible" is a Java concept different from the subsumption rule (an expression of a given type also has all the supertypes), which *does not* hold in Java; it means that in certain circumstances (specified in JLS Chapter 5) an expression of a certain type can be used in a context where another type is expected. In particular, there is a *widening reference conversion* which can take place, for instance, from a type which is a class to a superclass or to an implemented interface, and what happens in Jam is that this kind of conversion also takes place from a type which is a mixin instance to the corresponding mixin type and to the parent class.

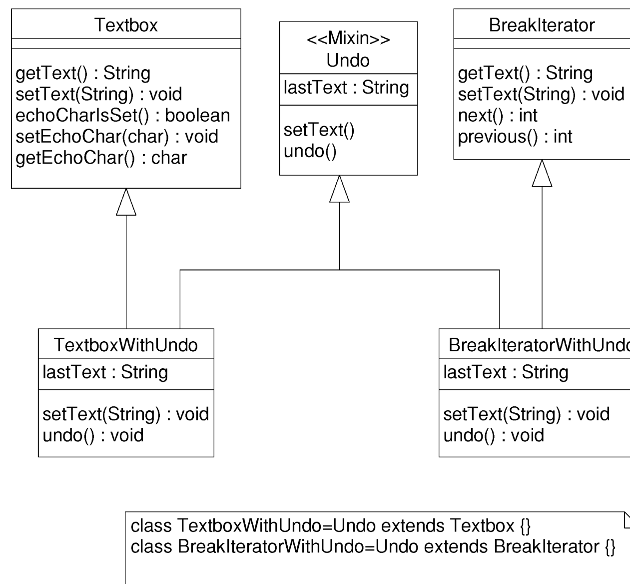

Fig. 4. Type hierarchy for the example.

Furthermore, mixins are handy to implement adapters [Gamma et al. 1995], that is, adapting an interface of a class to another interface that a client expects in order to make them work together. A mixin can inherit the interface of a class (not to be confused with a Java interface) providing a whole other interface, the one that a client expects. Such a mixin can be plugged in at various points in a class hierarchy, providing a more general approach than the usual wrapper based on a class.

Finally, note that in Jam a mixin cannot be used for creating objects; in other words, expressions of the form `new M()`, where `M` is a mixin, are not correct. As a consequence, a mixin type is always used as a static type and never as a dynamic type.

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, besides declaring and inheriting nonabstract instance methods and declaring (instance) fields, a usual heir class can also

—implement interfaces,
—declare constructors,
—inherit (instance) fields,
—inherit static fields,
—inherit static methods,
—declare static fields,
—declare static methods,

—inherit abstract methods,

—declare abstract methods.

Following the design principle that a mixin should be as similar as possible to a usual heir class, mixins should provide all these features. Later on we illustrate each of them in detail, highlighting and justifying some restrictions. We also discuss some subtle points concerning (instance) field accesses from references of a mixin type.

2.2.1 *Interfaces.* A mixin can implement interfaces in exactly the same way a class does.

2.2.2 *Constructors.* In Jam, constructors cannot be declared in a mixin, but for each single mixin instance, they can be declared 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.

2.2.3 *Inherited Instance Fields.* In a mixin it is possible to access inherited (instance) fields in the same way 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).

2.2.4 *Static Members.* Although in Jam static components are declared in the same way as instance components except, of course, in 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 by instantiating `M`, invocations `H.m()` or *e*`.m()`[7] with *e* of type `H` are legal. The same rule holds for fields: `f` can be accessed only by expressions like `H.f` or *e*`.f` with *e* of type `H`. 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;

—leaving to the programmer (introducing a new keyword, or analogous mechanisms) the decision whether a component should be shared between all the

[7]We maintain this alternative syntax for compatibility reasons only; see JLS 15.11.1.

mixin instances or not. In this scenario accessing static components through the mixin type would be allowed for shared components only, while it would remain forbidden for the others. This choice would require the addition of some static constraints like, for instance, that a shared static method cannot invoke an unshared static one.

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 cannot belong to mixin types in any case) but, like in Java, static methods cannot be abstract.

2.2.5 *Abstract Methods.* In a mixin it is possible to declare abstract methods (that is, methods without a body). For each of them the method, with the specified type, will be part of the mixin type. A class obtained by instantiating a mixin that contains abstract methods must necessarily be declared abstract (vice versa, like in Java, see JLS 8.1.1.1, it is allowed to declare an abstract class without any abstract method).

Note that declaring an inherited abstract method, as in the following code:

```
mixin M {
  inherited abstract void m();
}
```

is a request for the parent class to declare, but not necessarily to implement, the method; that is, the method `m()` in the parent class *can* be abstract. As a consequence, it is not possible to invoke `m()` via `super` in the mixin body. On the contrary, if the method is declared `inherited`, as in the following code:

```
mixin M {
  inherited void m();
}
```

then it is possible to invoke `m()` via `super` in the mixin body. In this case, if a class on which we try to instantiate the mixin declares abstract such a method, then the instantiation is not correct.

In a mixin it is also allowed to redefine as `abstract` an inherited method, as in the following code:

```
mixin M {
  inherited void m();
  abstract void m();
}
```

In this case the meaning is that the parent class should provide an implementation for the method `m`, which will become abstract from this point of the hierarchy on (thus it is left to a further subclass to provide an implementation). We have chosen to allow this use of the `abstract` modifier since the same is allowed in Java (JLS 8.4.3.1) for usual heir classes. Obviously all instances of `M` have to be declared abstract.

2.2.6 *Ambiguity in Field Accesses in Case of Double Mixin Instantiation.* If we apply two times to a class, say `Base`, a mixin which defines a field, say

```
mixin M {int f;}
class Once = M extends Base
class Twice = M extends Once
```

then an instance of `Twice` has two `f` fields, each from a different application of `M`. Indeed following copy semantics this should be equivalent to

```
class Once = extends Base {int f;}
class Twice = extends Once {int f;}
```

In a field access `twice.f` with `twice` of static type `Twice`, it is clear that the field corresponding to the second application is selected (the previous is hidden), while in an analogous field access `once.f` with `once` of static type `Once` the field corresponding to the first application is selected. The problem is what should happen with a field access `m.f` where `m` is of static type `M`. Note that in this case the copy principle does not help in finding a "natural" semantics since the fact that `M` can be used as a type has no correspondence in copy semantics.

In order to avoid this problem, we could have forbidden `m.f` with `m` of a mixin type, that is, we could have *not* considered fields as part of the mixin type, exactly as we do for static members. However, we have preferred to keep a feature which can be useful from the methodological point of view (the possibility of writing code which selects a field uniformly for all mixin instances) and in the "normal" case (no multiple mixin instantiation) not ambiguous.

The semantics which is given by the translation in Java defined in Section 4 is that, in case of ambiguity, the field which is selected is determined by the dynamic type of `m` (that is, it is that corresponding to the first instantiation if the dynamic type of `m` is `Once`, while it is that corresponding to the second instantiation if the dynamic type is `Twice`). However, in case of multiple instantiations of a mixin type which declares fields, the Jam compiler produces a warning to make the user aware that field accesses through a reference of the mixin type have nonobvious semantics.

Another possibility was to consider instance fields coming from multiple mixin instantiations as "shared" (that is, in the example an instance of `Twice` has only one `f` field). However, analogously to what we have done in the case of static members, we have discarded this choice since it does not correspond to copy semantics.

Finally, a different semantics could have been obtained by introducing a mechanism of "views" similar to what happens in C++ for obtaining that, in situations like

```
Once o = new Twice();
M m = o;
m.f;
```

the selected field is that corresponding to the first instantiation and not that corresponding to the second, as happens with our semantics.

In this way the semantics of the assignment `M m = o` would not be simply that `m` refers now to the object created in the first line of code (as it is in Java semantics), but that `m` refers now to a "view" of type `Once` which is obtained from this object by forgetting some part of it. We have discarded this solution because it is not in the spirit of the Jam (and Java) design, which is to preserve the simple semantics mentioned above.

2.3 Constraints on Instantiation

As mentioned in Section 2.1, 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`. We briefly illustrate the different interference cases and at the end discuss our overall design choice.

—*Illegal overriding/hiding*. A method in `P` is illegally overridden (hidden) (JLS 8.4.6.3) by a defined method in `M` if it has the same signature (name and arguments type) but different return type, or different kind (`instance` or `static`) or incompatible `throws` clause. This would produce an illegal Java class, and hence is forbidden in Jam.

—*Unexpected overriding*. A method in `P` is incidentally overridden by a method defined in `M` if it has the same signature (name and arguments type), return type, kind, and a compatible (JLS 8.4.4) `throws` clause. For instance, instantiating the mixin `Undo` on a class with a `void undo()` method produces an unexpected overriding. This situation would produce a correct Java class, but looks somehow undesirable, since there is some either overriding or hiding which was not planned when declaring the mixin; in particular, an invocation of the method in the parent would refer, in the heir, to a different method, and hence is forbidden in Jam. Note that unexpected hiding of fields does not cause an analogous problem since in that case the binding is static. Unexpected hiding of static methods is somehow problematic: on the one hand, considering that the binding is static, it should not cause any harm but, on the other hand, the translation into plain Java forces compliance with the rules on overriding/hiding. So we allow the hiding of a static method only if the new method respects the rules on hiding, that is, has the same return type and a throws clause that is compatible with the one of the method it hides.

—*Ambiguous overloading*. There exist contexts in which the presence of the method in `P` makes ambiguous, with respect to 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 we 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, and hence both methods are applicable and neither is more specific (JLS 15.12.2.2).

In general, if two methods have the same name, then the addition of one may make ambiguous, with respect to 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). 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. For an alternative approach in which the constraints are implicit, see, for instance, C++ templates, where type checks are all performed at instantiation time.

We have chosen to avoid these three kinds of interferences forbidding all the instantiations on parent classes having some extra-method (that is, not corresponding to an `inherited` method) with the same name of some (either inherited or declared) method in the mixin and interfering arguments type (see Figure 18 for the formal definition).[8] We briefly comment on this choice (for the interested reader, an extended discussion on different possibilities in handling interferences when extending "external" parent classes is given in Section 3 of Ancona and Zucca [2001]).

The problem of interferences between the mixin and the actual parent class is a particular case of the situation in which two software modules are composed, and conflict may arise among components which have the same name in both (and for which no precedence is indicated, as happens in Jam for components specified as `inherited`). The same kind of conflicts arises, for example, in multiple inheritance.

A possible solution is to allow conflict, and to complicate dynamic semantics by means of some "view" mechanism. This is done, for instance, in multiple inheritance in C++, in the Java-like mixin language MIXEDJAVA [Flatt et al. 1998] discussed in Section 6.1, in some existing proposals for extending Java-like languages with a module system [Ancona and Zucca 2001; Fisher and Reppy 1999]. Note that the "view" mechanism can either work in only one direction giving priority to the heir (like in overriding), that is, the component in the parent is hidden by the component in the heir, and can be retrieved by inserting a cast, as it is in [Ancona and Zucca 2001; Fisher and Reppy 1999], or in both directions with no priority. In the latter case both components must be retrieved by a cast; otherwise there is ambiguity, as there is in multiple inheritance and in Flatt et al. [1998].

[8] The only exception is the case of static methods with exactly the same signature, that is, unexpected hiding, which is allowed.

```
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. 5. Overloading conflict between inherited and defined methods.

An alternative direction is to keep dynamic semantics simple (standard Java semantics in our case). In the case of mixins, this can be achieved by either interpreting conflict as overriding (that is, accepting what we called "unexpected overriding"), or by complicating the type system forbidding conflicts. In designing Jam we preferred the former.

Different choices (for instance, hiding these extra-methods, as done, e.g., in Ancona and Zucca [2001] and Fisher and Reppy [1999]) would violate our copy principle and a translation in Java would hardly be possible. See Section 6.1 for some further discussion on this point.

2.4 Overloading

The Java rules for overloading resolution (JLS 15.12.2) 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 (and hence, can be converted to) the corresponding mixin type. However, some special care is needed in handling the situation when there is an overloading conflict between either an inherited and a defined method or two inherited methods in a mixin.

We illustrate the first problem in term of the following example.

In the first part of the code shown in Figure 5, `B` is a subtype of `A` and `Heir` is a subtype of `Parent`. The class `Parent` defines a method named `f` with one

```
class Parent {}
class Heir extends Parent {}

class C {
   void f(Heir h) {}
}

class D extends C {
   void f(Parent p) {}
}
```

```
mixin M {
   inherited void f(Parent p) ;
   inherited void f(Heir h) ;
   void foo(Heir h) {
      f(h) ;
   }
}

class Broken = M extends D {}
```

Fig. 6. Overloading conflict among inherited methods.

argument of type B, while its subclass Heir defines another method with the same name and argument's type A. Consider an invocation h.f(b) with h of type Heir and b of type B. There are two applicable methods, the first having a more specific parameter type than the second, *but* is declared in a parent (and hence less specific) class. Java rules for determining when a method is more specific than another (JLS 15.12.2.2, formalized later on in Figure 30) compare *both* the class containing the declaration and the parameter types, in this case, the pairs <Parent, B> and <Heir, A>; neither is more specific, and hence the invocation is considered ambiguous.

If we now consider the declaration of the mixin M, and an invocation m.f(b) with m of type M and b of type B, then the situation is analogous to the preceding: there are two applicable methods, the first having more specific parameter type than the latter, *but* it is inherited, that is, required to be provided from a parent class, while the second is defined in the mixin; hence, we expect that the invocation should be ambiguous as well.

In other words, we consider that inherited methods in a mixin M are declared "at a less specific level" with respect to defined methods. In order to achieve this, referring to the example, we model the two applicable methods by the two pairs <*Parent*(M), B> and <M, A> where in the first pair *Parent*(M) means that the method has been declared in the mixin M as inherited method. See Section 3.6 for the precise formal definitions.

The second problem concerns overloading among inherited methods. We forbid this possibility, in the case of two methods such that the arguments type of the former is a subtype of the arguments type of the latter, since this kind of overloading may introduce ambiguities on method calls when the inherited methods are not provided by the same class. Consider, for instance, the code shown in Figure 6. The method invocation f(h) in foo is ambiguous because there are two applicable methods (C.f(Heir) and D.f(Parent)) and neither is more specific.[9]

[9] D is more specific than C, but Parent is less specific than Heir.

```
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. 7. Problem in using `this` in mixins.

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 (respectively 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 (JLS 15.8.3). 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 instance `H` of a mixin `M` the expression `this` has static type `H`, as 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 (the Java class `H` corresponding to) a mixin instance (following the copy principle) to always be a well-formed Java class, since in Java `this` has type `H` in this class. In most cases this is not a problem because `H` is a subtype of `M` and therefore the former can be used in place of the latter. Nevertheless, this can lead to unsound situations in some subtle cases involving overloading. Let us consider the example in Figure 7.

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

Now, 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. But in this declaration both methods are applicable to the invocation `A.f(this)`, and the second is more specific; hence the invocation has type `boolean` and cannot be used for initializing the variable `i`.

In order to avoid these situations, we have taken for Jam a quite drastic design decision, that is, forbidding the use of `this` as argument in method and constructor invocation inside a mixin. However, note that this rule is not so

restrictive as it may look; for instance, the problem outlined in Figure 7 can be solved by replacing `this` with a local variable `m` of type `M`:

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

2.6 Limitations

As mentioned in Section 2.1, mixins in Jam are "parametric heir classes" in the sense that each instantiation of a mixin `M` on a parent, say `P`, corresponds to extend `P` by the same definitions, those given in `M`; however, they are *not* parametric with respect to the types of the parent and heir classes. In other words, in a mixin there is no way to refer either to the "generic" parent class on which the mixin will be instantiated or to the "generic" heir class obtained by instantiation; it is possible to refer to the mixin type `M`, but this type will remain `M` in every instantiation.

As a consequence, a Java declaration of a class `H` which extends `P` which contains references to either `H` or `C` cannot be "abstracted" in a mixin declaration. Consider for instance the following simple example:

```
class P {}
class H extends P {
  public H m() {return this;}
}
```

We would be tempted to "abstract" the above in a mixin declaration as follows;

```
mixin M {
  public M m() {return this;}
}
```

However, instantiating this mixin on `P`

```
class H = M extends P
```

the class we obtain *is not equivalent* to the original class but, by copy principle, to the following declaration:

```
class H extends P {
  public M m() {return this;}
}
```

Even worse, if there is in the heir class some reference to the parent type, as in

```
class H extends P {
  public P m() {return this;}
}
```

then we cannot write an analogous mixin definition since there is no way to refer to the "generic" parent type inside.

In order to do this, we should introduce canonical notations for the parametric names of the parent and heir class, say P* and H*, respectively, as shown below:

```
mixin M {
  public H* m() {return this;}
  public P* m() {return this;}
}
```

However, in this way we would loose one of the two design principles of Jam, that is, the fact that a mixin name can be used as a type. See Section 6.2 for a further discussion on this point.

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 (except the fact that `mixin` and `inherited` are keywords in Jam); an extension to Java 1.1 would require some more work at the implementation level, but it 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 only considers a subset of the language chosen to be a minimal but sufficient set for analyzing 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 to 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 Drossopoulou and Eisenbach [1999]: constructors, `static` members, checked exceptions,[10] `abstract` classes and methods, method invocations, and field accesses via `super`. An extended presentation of the formal model for exceptions types used here can be found in Ancona et al. [2001].

The structure of this section is the following. After introducing some basic notations in Section 3.1, we define the Jam abstract syntax in Section 3.2. Then, we define the Jam type system, which formally consists of a set of metarules which define the validity of different kinds of *judgments* which, under some type assumptions modeled by *environments*, assign types to the different constructs of the language. We first define the different kinds of types (Section 3.3) and the environments (Section 3.4), and then give the metarules which define the validity of judgments. The presentation of these metarules is split into Sections 3.5 and 3.6, corresponding to the fact that the well-formedness of a Jam program

[10] Checked exceptions have been considered in a recent improved version [Drossopoulou et al. 1999].

(collection of class, interface, and mixin declarations) is established in two steps: first, the well-formedness of the environment which contains the type information extracted from the program (roughly, the program deprived of constructor and method bodies) is established (in this phase, for instance, absence of cycles in the inheritance hierarchy and rules on overriding methods are checked); then, the well-formedness of constructor and method bodies with respect to the environment is established.

3.1 Notations

We use the `typewriter` style for terminal and *italic* for nonterminal symbols. The terminals `iname`, `cname`, and `mxname` indicate interface, class, and mixin names, respectively, whereas `name` indicates either method or field or parameter names. 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 nonempty set of occurrences of A.

Angle brackets $<$ and $>$ are used as metaparentheses.

3.2 Abstract Syntax

Figure 8 shows the Jam abstract syntax; the LALR grammar used in the implementation can be found at: `http://www.disi.unige.it/person/LagorioG/jam`.

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

A program is a set of interface, class, and mixin declarations. Simple types are divided in two categories: primitive types (here we consider only `int` and `boolean`) and reference types. Reference types can be, besides class and interface names, mixin names (first Jam-specific production). Method return types (*ret-type*) are simple types or `void`. Exceptions types are sets of class names.

A class declaration consists of the optional `abstract` modifier, followed by the keyword `class`, the name of the class, the name of the superclass, the set of the implemented interfaces, and the constructor, field, and method declarations. 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 (second Jam-specific production).

An interface declaration consists of the keyword `interface`, followed by the name of the interface, the set of the extended interfaces, and a set of (necessarily `abstract`) methods declarations.

A mixin declaration (third Jam-specific production) logically consists of two parts: the former contains the declarations of the defined components,

```
           ref-type ::= mxname
               cdecl ::= [ abstract ] class cname = mxname extends
                         cname {constructor⊛}
               mdecl ::= mixin mxname implements iname⊛
                         { ⟨ [inherited] field ⟩⊛ ⟨ [inherited] meth ⟩⊛ }

                prog ::= decl⊛
                decl ::= idecl | cdecl | mdecl
         simple-type ::= prim-type | ref-type
            ref-type ::= iname | cname
           prim-type ::= int | boolean
            ret-type ::= simple-type | void
            exc-type ::= cname*
               cdecl ::= [ abstract ] class cname extends cname
                         implements iname⊛
                         { constructor⊛ field⊛ meth⊛ }
               idecl ::= interface iname extends iname⊛ { imeth⊛ }
               imeth ::= abstract ret-type name params throws exc-type ;
              params ::= (⟨ simple-type name ⟩*)
         constructor ::= cname params throws exc-type
                         { super(expr*) ; stmts }
                meth ::= [ static ] ret-type name params
                         throws exc-type mbody |
                         imeth
               mbody ::= { stmts return [expr] ;}
               field ::= [ static ] simple-type name ;
               stmts ::= stmt*
                stmt ::= s-expr ; | if (expr) stmt else stmt |
                         { stmts } | throw expr ; |
                         try { stmts } ⟨ catch cname name { stmts } ⟩⊛
                         finally { stmts }
              s-expr ::= assignment | method-invocation | new-expr
          assignment ::= name = expr | primary.name = expr |
                         super.name = expr
   method-invocation ::= primary.name(expr*) | super.name(expr*) |
                         name(expr*)
            new-expr ::= new cname(expr*)
             primary ::= name | null | this | (expr) | new-expr |
                         primary.name | super.name | method-invocation
                expr ::= primary | assignment | literal | ...
```

Fig. 8. Jam abstract syntax.

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 labeled with the `inherited` modifier. Moreover, the set of the implemented interfaces is specified.

Note that a constructor body always begins with a parent class constructor explicit invocation (using `super`); we have not considered the invocation of a constructor of the same class (using `this`) since such invocations are simply syntactic shortcuts (recursive invocations are not allowed (JLS 8.8.5)).

type	::=	*ref-type* | *prim-type* | `nil`
field-type	::=	*field-kind simple-type*
field-kind	::=	`instance` | `static`
args-type	::=	*simple-type**
constr-type	::=	*args-type* `throws` *exc-type*
meth-type	::=	*meth-kind ret-type* `throws` *exc-type*
meth-sig	::=	`name` *args-type*
meth-kind	::=	`instance` | `abstract` | `static`
fields-type	::=	$\langle$`name` : *field-type*$\rangle^{\circledast}$
meths-type	::=	$\langle$*meth-sig* : *meth-type*$\rangle^{\circledast}$
constrs-type	::=	*constr-type*$^{\oplus}$
module-type	::=	*fields-type meths-type*
class-type	::=	*class-kind constrs-type module-type*
class-kind	::=	`abstract` | `concrete`
interface-type	::=	*meths-type*
mixin-type	::=	*module-type* `inherited` *module-type*

Fig. 9. Jam types.

3.3 Types

In Figure 9 are defined the Jam types. A generic *type* can be a reference type, a primitive type (both defined in Figure 8) 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 Figure 8). A method type consists of the (method) kind (indicating whether the method is `instance`, `static` or `abstract`), 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. 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. The type *constrs-type* is a nonempty 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 (class) kind (indicating whether the class is `abstract` or `concrete`), a set of constructors, and a module type. 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.

3.3.1 *Valid Types.* A fields type is *valid* if field names are distinct, and, analogously, a methods type is *valid* if method signatures are distinct. In the following we will consider only valid fields and methods types, and use the following notations: if $FST = \{f_1 : FT_1, \ldots, f_n : FT_n\}$ is a valid fields type, then

env	::=	*basic-type-assertion*$^{\circledast}$
basic-type-assertion	::=	`cname` is_c *class-type* |
		`cname` $<_c^1$ `cname` |
		`name` $\lhd_i^1$ `iname` |
		`iname` is_i *interface-type* |
		`iname` $<_i^1$ `iname` |
		`mxname` is_m *mixin-type* |
		`cname` $\lhd_m$ `mxname`
body-decl	::=	`class cname` { *constructor*$^{\circledast}$ *meth*$^{\circledast}$ } |
		`mixin mxname` { *meth*$^{\circledast}$ }
body-decls	::=	*body-decl*$^{\circledast}$
modname	::=	`cname` | `iname` | `mxname`

Fig. 10. Environments and body declarations.

$Dom(FST) = \{f_1, \ldots, f_n\}$ and $FST(f) = FT_i$ if $f = f_i$ for some $i \in \{1, \ldots, n\}$, $\perp$ otherwise; analogously for methods types.

3.4 Environments

A Jam program contains both type information and information needed at runtime (that is, the constructor and method bodies). To simplify the formal definition, following the approach used in Drossopoulou and Eisenbach [1999], we consider two components that can be extracted in a trivial way from a program: the environment Γ, which contains the type information, and the remaining part of the program, consisting of 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 Figure 10. 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:

$$\Gamma \vdash \diamond$$
$$\Gamma \vdash \{BD_1, \ldots, BD_n\}\diamond_{\text{Body-Decls}}$$

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 with respect to the type information in Γ. The validity of these two judgments is inductively defined introducing other judgments relative to subcomponents.

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$_i$ IT : the interface I declares the methods specified in IT (recall that interface and methods types coincide);

—$I <_i^1 I'$: 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.

For instance, the environment extracted from a program consisting in the declaration of the mixin `Undo`, given in Figure 1, and its instantiation in Figure 2 is the following:

$$
\begin{array}{rll}
\Gamma = \{ \quad \texttt{Undo} & \text{is}_m & \{\texttt{lastText} : \texttt{instance String}\} \\
& & \{\texttt{setText}, \texttt{String} : \texttt{instance void throws } \emptyset, \\
& & \texttt{undo}, \epsilon : \texttt{instance void throws } \emptyset\} \\
& & \texttt{inherited } \emptyset \\
& & \{\texttt{setText}, \texttt{String} : \texttt{instance void throws } \emptyset, \\
& & \texttt{getText}, \epsilon : \texttt{instance String throws } \emptyset\}, \\
\texttt{Component} & \text{is}_c & \ldots, \\
\texttt{Textbox} & \text{is}_c & \ldots, \\
\texttt{Textbox} & <_c^1 & \texttt{Component}, \\
\texttt{TextboxWithUndo} & \text{is}_c & \texttt{concrete } \{\epsilon \texttt{ throws } \emptyset\}\ \emptyset\ \emptyset, \\
\texttt{TextboxWithUndo} & <_c^1 & \texttt{Textbox}, \\
\texttt{TextboxWithUndo} & \lhd_m & \texttt{Undo} \\
\} \cup \Gamma_{std} & &
\end{array}
$$

We have denoted by Γ_{std} the environment corresponding to the declarations of the standard classes like `Object`, `Throwable`, and, in general, all those belonging to the package `java.lang`.

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 Γ, that is, $C \in Classes(\Gamma)$ iff C is$_c$ $CT \in \Gamma$;

—$StandardClasses(\Gamma)$ the set of all class names defined in Γ which are not mixin instances, that is, $C \in Classes(\Gamma)$ and $\nexists M\ such that\ C \lhd_m M \in \Gamma$;

—$Interfaces(\Gamma)$ the set of all interface names defined in Γ, that is, $I \in Interfaces(\Gamma)$ iff I is$_i$ $IT \in \Gamma$;

—$Mixins(\Gamma)$ the set of all mixin names defined in Γ, that is, $M \in Mixins(\Gamma)$ iff M is$_m$ $MKT \in \Gamma$.

$$
\begin{array}{rcl}
\textit{judgment} & ::= & \textit{env} \vdash \textit{type-assertion} \\
\textit{type-assertion} & ::= & \textit{basic-type-assertion} \mid \\
& & \texttt{cname} \leq_c \texttt{cname} \mid \texttt{iname} \leq_i \texttt{iname} \mid \\
& & \texttt{name} \lhd_i \texttt{iname} \mid \textit{modname} \leq \textit{modname} \mid \\
& & \textit{exc-type} \leq_e \textit{exc-type} \mid \textit{fields-type} \leq_{fields} \textit{fields-type} \mid \\
& & \textit{meth-type} \leq_{meth} \textit{meth-type} \mid \textit{meths-type} \leq_{meths} \textit{meths-type} \mid \\
& & \textit{module-type} \leq_{mod} \textit{module-type} \mid \\
& & \textit{ref-type} \diamond_{\mathrm{RefType}} \mid \textit{type} \diamond_{\mathrm{Type}} \mid \\
& & \textit{prim-type} \diamond_{\mathrm{PrimType}} \mid \textit{simple-type} \diamond_{\mathrm{SimpleType}} \mid \\
& & \textit{ret-type} \diamond_{\mathrm{RetType}} \mid \textit{field-type} \diamond_{\mathrm{FieldType}} \mid \\
& & \textit{exc-type} \diamond_{\mathrm{ExcType}} \mid \textit{args-type} \diamond_{\mathrm{ArgsType}} \mid \\
& & \textit{constr-type} \diamond_{\mathrm{ConstrType}} \mid \textit{meth-type} \diamond_{\mathrm{MethType}} \mid \\
& & \textit{constrs-type} \diamond_{\mathrm{ConstrsType}} \mid \textit{fields-type} \diamond_{\mathrm{FieldsType}} \mid \\
& & \textit{meths-type} \diamond_{\mathrm{MethsType}} \mid \textit{module-type} \diamond_{\mathrm{ModuleType}} \mid \\
& & \textit{class-type} \diamond_{\mathrm{ClassType}} \mid \textit{interface-type} \diamond_{\mathrm{InterfaceType}} \mid \\
& & \textit{mixin-type} \diamond_{\mathrm{MixinType}} \mid \textit{modname} : \textit{fields-type}\ \textit{meths-type}
\end{array}
$$

Fig. 11. Judgments I.

3.5 Typing Rules I

In this subsection, we give the first part of the metarules of the Jam type system, that is, those related to environments.Metarules related to features introduced by Jam, that is, not belonging to the Java type system, are distinguished by the label Jam. Moreover, some metarules are labeled by a different label—[Jam]—and have some side condition inside square brackets, denoting optionality; in this case, the intended meaning is that the metarule without the optional side conditions belongs to the Java type system. The judgments which appear in these metarules have generic form $\Gamma \vdash \gamma$ with Γ environment and γ *type-assertion*. The syntax of the judgments used in this section is given in Figure 11. The type assertions have the following informal meaning:

—$C \leq_c C'$: the class C is a subclass of the class C';
—$I \leq_i I'$: the interface I is a subinterface of the interface I';
—$T \lhd_i I$: the module (either class or mixin) T implements the interface I;
—$T \leq T'$: the type T is a subtype of the type T';
—$ET_1 \leq_e ET_2$: the exceptions type ET_1 is a subtype of the exceptions type ET_2;
—$FST_1 \leq_{fields} FST_2$: the fields type FST_1 is a subtype of the fields type FST_2;
—$MT_1 \leq_{meth} MT_2$: the method type MT_1 is a subtype of the method type MT_2;
—$MST_1 \leq_{meths} MST_2$: the methods type MST_1 is a subtype of the methods type MST_2;
—$MOD_1 \leq_{mod} MOD_2$: the module type MOD_1 is a subtype of the module type MOD_2;
—$T \diamond_{\mathrm{RefType}}$: T is a well-formed reference type;
—$T \diamond_{\mathrm{Type}}$: T is a well-formed type;
—$T \diamond_{\mathrm{PrimType}}$: T is a well-formed primitive type;

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

Fig. 12. Basic type assertion.

—$T \diamond_{\text{SimpleType}}$: T is a well-formed simple type;
—$T \diamond_{\text{RetType}}$: T is a well-formed return type;
—$FT \diamond_{\text{FieldType}}$: FT is a well-formed field type;
—$ET \diamond_{\text{ExcType}}$: ET is a well-formed exceptions type;
—$AT \diamond_{\text{ArgsType}}$: AT is a well-formed arguments type;
—$msig \diamond_{\text{MethSign}}$: $msig$ is a well-formed method signature;
—$KT \diamond_{\text{ConstrType}}$: KT is a well-formed constructor type;
—$MT \diamond_{\text{MethType}}$: MT is a well-formed method type;
—$KST \diamond_{\text{ConstrsType}}$: KST is a well-formed constructors type;
—$FST \diamond_{\text{FieldsType}}$: FST is a well-formed fields type;
—$MST \diamond_{\text{MethsType}}$: MST is a well-formed methods type;
—$MODT \diamond_{\text{ModuleType}}$: $MODT$ is a well-formed module type;
—$CT \diamond_{\text{ClassType}}$: CT is a well-formed class type;
—$IT \diamond_{\text{InterfaceType}}$: IT is a well-formed interface type;
—$MXT \diamond_{\text{MixinType}}$: MXT is a well-formed mixin type;
—T:$MODT$: the module (either class, or interface, or mixin) T has type $MODT$.

3.5.1 *Basis.* The metarule in Figure 12 provides the basis for the inductive definition of the validity of judgments.

3.5.2 *Relations Between Types.* The metarules in Figure 13 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 $<_c^1$ 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 $\lhd_m$ and the subclass and subinterface relations. The new relation introduced in Jam is the *instantiation* relation, denoted by $\lhd_m$, 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 Figure 14 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 (23) stands for $\Gamma \vdash ET \leq_e ET'$ and $\Gamma \vdash ET' \leq_e ET$, that is, every exception in one type must be subtype of an exception

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

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

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

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

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

$$(12)\quad \frac{\Gamma \ \vdash \ C \leq_c C'}{\Gamma \vdash C \leq C'} \qquad (13)\quad \frac{\Gamma \ \vdash \ I \leq_i I'}{\Gamma \vdash I \leq I'}$$

$$(14\text{-Jam})\quad \frac{\Gamma \ \vdash \ C \lhd_m M}{\Gamma \vdash C \leq M} \qquad (15)\quad \frac{\Gamma \ \vdash \ T \leq T}{\Gamma \vdash T \leq \text{Object}}$$

$$(16)\quad \frac{\Gamma \ \vdash \ T \leq T}{\Gamma \vdash \text{nil} \leq T} \qquad (17)\quad \frac{\Gamma \ \vdash \ T \lhd_i I}{\Gamma \vdash T \leq I}$$

$$(18\text{-Jam})\quad \frac{\Gamma \ \vdash \ M \text{ is}_m\ MXT}{\Gamma \vdash M \leq M} \qquad (19)\quad \frac{\Gamma \vdash T \leq T' \quad \Gamma \vdash T' \leq T''}{\Gamma \vdash \ T \leq T''}$$

$$(20)\quad \frac{}{\begin{array}{l}\Gamma \vdash \texttt{void} \leq \texttt{void} \\ \Gamma \vdash \texttt{int} \leq \texttt{int} \\ \Gamma \vdash \texttt{boolean} \leq \texttt{boolean}\end{array}}$$

Fig. 13. Subclass, subinterface, implementation, and widening relations.

in the other, and conversely). Note that, in (21), it is possible that $E'_i = E'_j$ holds for some i, j.

3.5.3 *Well-Formedness of Types.* The metarules in Figures 15 and 16 all express the factthat some Jam type is well-formed. They use the following auxiliary notations:

—for field types:
 —$Kind(KT) = \mathrm{K}$,
 —$Type(KT) = \mathrm{T}$;
—for constructor types:
 —$Args(AT$ `throws` $ET) = AT$,
 —$Exc(AT$ `throws` $ET) = ET$;
—for method signatures:
 —$Name(M\ AT) = M$,
 —$Args(M\ AT) = AT$;

$$(21)\quad \frac{\Gamma \vdash E_1 \leq_c E'_1 \ldots \Gamma \vdash E_n \leq_c E'_n \quad \Gamma \vdash \{E_1, \ldots, E_n\} \diamond_{\text{ExcType}} \quad \Gamma \vdash \{E'_1, \ldots, E'_m\} \diamond_{\text{ExcType}}}{\Gamma \vdash \{E_1, \ldots, E_n\} \leq_e \{E'_1, \ldots, E'_m\}} \quad 0 \leq n \leq m$$

$$(22)\quad \frac{\Gamma \vdash \{f_1 : FT_1, \ldots, f_n : FT_n\} \diamond_{\text{FieldsType}}}{\Gamma \vdash \{f_1 : FT_1, \ldots, f_n : FT_n\} \leq_{fields} \{f_1 : FT_1, \ldots, f_m : FT_m\}} \quad m \leq n$$

$$(23)\quad \frac{\Gamma \vdash RT \diamond_{\text{RetType}} \quad \Gamma \vdash ET =_e ET'}{\Gamma \vdash \ K\ RT\ \texttt{throws}\ ET \leq_{meth}\ K\ RT\ \texttt{throws}\ ET'}$$

$$(24)\quad \frac{\Gamma \vdash \{msig_1 : MT_1, \ldots, msig_n : MT_n\} \diamond_{\text{MethsType}} \quad \Gamma \vdash \{msig_1 : MT'_1, \ldots, msig_k : MT'_k\} \diamond_{\text{MethsType}} \quad \Gamma \vdash MT_1 \leq_{meth} MT'_1 \ \ldots\ \Gamma \vdash MT_k \leq_{meth} MT'_k}{\Gamma \vdash \ \{msig_1 : MT_1, \ldots, msig_n : MT_n\} \leq_{meths} \{msig_1 : MT'_1, \ldots, msig_k : MT'_k\}} \quad k \leq n$$

$$(25)\quad \frac{\Gamma \vdash FST \leq_{fields} FST' \quad \Gamma \vdash MST \leq_{meths} MST'}{\Gamma \vdash FST\ MST \leq_{mod} FST'\ MST'}$$

Fig. 14. Relations on exceptions, fields, methods, and module types.

—for method types:
- —$Kind(K\ RT\ \texttt{throws}\ ET) = K$,
- —$Ret(K\ RT\ \texttt{throws}\ ET) = RT$,
- —$Exc(K\ RT\ \texttt{throws}\ ET) = ET$;

—for sets of methods:
- — $Kind(\{msig_1 : MT_1, \ldots, msig_k : MT_k\}) = \begin{cases} \texttt{abstract} & \text{if } \exists i \text{ s.t. } Kind(MT_i) = \texttt{abstract}, \\ \texttt{concrete} & \text{otherwise;} \end{cases}$

—for module types:
- —$Kind(FST\ MST) = Kind(MST)$;

—for class types:
- —$Kind(K\ KST\ FST\ MST) = K$.

We briefly illustrate the metarules in Figure 16. A Jam type is well-formed with respect to an environment Γ if its components are well-formed with respect to Γ and, moreover, the following conditions hold:

(39) In a constructor type there are no constructors with the same signature (arguments type) (JLS 8.8.2).

(40) In a fields type there are no fields with the same name (JLS 8.3), that is, the fields type is valid.

(41) In a methods type there are no methods with the same signature (JLS 8.4), that is, the methods type is valid.

(42) In an interface type all methods are abstract (JLS 9.4).

(43) In a mixin type the defined module type is compatible with the inherited one, in a sense formalized by the update operation on methods types $_[_]_\Gamma$ defined in the sequel.

(26) $$\frac{\Gamma \vdash C \text{ is}_c\ CT}{\Gamma \vdash C \diamond_{\text{RefType}}}$$

(27) $$\frac{\Gamma \vdash I \text{ is}_i\ IT}{\Gamma \vdash I \diamond_{\text{RefType}}}$$

(28-Jam) $$\frac{\Gamma \vdash M \text{ is}_m\ MXT}{\Gamma \vdash M \diamond_{\text{RefType}}}$$

(29) $$\frac{}{\begin{array}{l}\Gamma \vdash \texttt{nil} \diamond_{\text{Type}} \\ \Gamma \vdash \texttt{int} \diamond_{\text{PrimType}} \\ \Gamma \vdash \texttt{boolean} \diamond_{\text{PrimType}} \\ \Gamma \vdash \texttt{void} \diamond_{\text{RetType}}\end{array}}$$

(30) $$\frac{\Gamma \vdash T \diamond_{\text{RefType}}}{\begin{array}{l}\Gamma \vdash T \diamond_{\text{SimpleType}} \\ \Gamma \vdash T \diamond_{\text{Type}}\end{array}}$$

(31) $$\frac{\Gamma \vdash T \diamond_{\text{PrimType}}}{\begin{array}{l}\Gamma \vdash T \diamond_{\text{SimpleType}} \\ \Gamma \vdash T \diamond_{\text{Type}}\end{array}}$$

(32) $$\frac{\Gamma \vdash T \diamond_{\text{SimpleType}}}{\Gamma \vdash T \diamond_{\text{RetType}}}$$

(33) $$\frac{\Gamma \vdash T \diamond_{\text{SimpleType}}}{\Gamma \vdash K\ T \diamond_{\text{FieldType}}}$$

(34) $$\frac{\Gamma \vdash C_1 \leq_c \texttt{Throwable} \ldots \Gamma \vdash C_n \leq_c \texttt{Throwable}}{\Gamma \vdash \{C_1, \ \ldots, \ C_n\} \diamond_{\text{ExcType}}} \quad n \geq 0$$

(35) $$\frac{\Gamma \vdash T_1 \diamond_{\text{SimpleType}} \ldots \Gamma \vdash T_n \diamond_{\text{SimpleType}}}{\Gamma \vdash T_1 \ \ldots \ T_n \diamond_{\text{ArgsType}}} \quad n \geq 0$$

(36) $$\frac{\Gamma \vdash AT \diamond_{\text{ArgsType}}}{\Gamma \vdash \texttt{id}\ AT \diamond_{\text{MethSign}}}$$

(37) $$\frac{\Gamma \vdash AT \diamond_{\text{ArgsType}}\ \Gamma \vdash ET \diamond_{\text{ExcType}}}{\Gamma \vdash AT\ \texttt{throws}\ ET \diamond_{\text{ConstrType}}}$$

(38) $$\frac{\Gamma \vdash RT \diamond_{\text{RetType}}\ \ \Gamma \vdash ET \diamond_{\text{ExcType}}}{\Gamma \vdash K\ RT\ \texttt{throws}\ ET \diamond_{\text{MethType}}}$$

Fig. 15. Well-formed types I.

3.5.4 *Type Assignments.* The metarules in Figure 17 define type assignments, that is, express the fact that some module (either class or interface or mixin) has a given type.

We use some auxiliary functions:

—*ParentInterfaces*(Γ, I) is the set of all interfaces directly extended by the interface I, that is, *ParentInterfaces*$(\Gamma, I) = \{I' | I <_i^1 I' \in \Gamma\}$.

—*DImplementedInterfaces*(Γ, T) is the set of all interfaces directly implemented by the either class or mixin T, that is, *DImplementedInterfaces* $(\Gamma, T) = \{I | T \lhd_i^1 I \in \Gamma\}$.

—*Abstract*(MST), *NonAbstract*(MST) are the methods types containing only the abstract (respectively nonabstract) methods of MST.

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

$$(39)\quad \frac{\Gamma \vdash KT_1 \diamond_{\text{ConstrType}} \;\dots\; \Gamma \vdash KT_n \diamond_{\text{ConstrType}}}{\Gamma \vdash \{KT_1, \dots, KT_n\} \diamond_{\text{ConstrsType}}} \quad \begin{array}{l} n > 0 \\ i \neq j \Rightarrow Args(KT_i) \neq Args(KT_j) \end{array}$$

$$(40)\quad \frac{\Gamma \vdash FT_1 \diamond_{\text{FieldType}} \;\dots\; \Gamma \vdash FT_n \diamond_{\text{FieldType}}}{\Gamma \vdash \{f_1 : FT_1, \dots, f_n : FT_n\} \diamond_{\text{FieldsType}}} \quad \begin{array}{l} n \geq 0 \\ i \neq j \Rightarrow f_i \neq f_j \end{array}$$

$$(41)\quad \frac{\begin{array}{c} \Gamma \vdash msig_1 \diamond_{\text{MethSign}} \;\dots\; \Gamma \vdash msig_n \diamond_{\text{MethSign}} \\ \Gamma \vdash MT_1 \diamond_{\text{MethType}} \;\dots\; \Gamma \vdash MT_n \diamond_{\text{MethType}} \end{array}}{\Gamma \vdash \{msig_1 : MT_1, \dots, msig_n : MT_n\} \diamond_{\text{MethsType}}} \quad \begin{array}{l} n \geq 0 \\ i \neq j \Rightarrow msig_i \neq msig_j \end{array}$$

$$(42)\quad \frac{\Gamma \vdash FST \diamond_{\text{FieldsType}} \;\; MST \diamond_{\text{MethsType}}}{\Gamma \vdash FST\ MST \diamond_{\text{ModuleType}}}$$

$$(43)\quad \frac{\Gamma \vdash MODT \diamond_{\text{ModuleType}} \;\; \Gamma \vdash KST \diamond_{\text{ConstrsType}}}{\Gamma \vdash K\ KST\ MODT \diamond_{\text{ClassType}}}$$

$$(44)\quad \frac{\Gamma \vdash \{msig_1 : MT_1 \dots msig_n : MT_n\} \diamond_{\text{MethsType}}}{\Gamma \vdash \{msig_1 : MT_1 \dots msig_n : MT_n\} \diamond_{\text{InterfaceType}}} \quad \begin{array}{l} \forall i \in \{1, \dots, n\} \\ Kind(MT_i) = \texttt{abstract} \end{array}$$

$$(45\text{-Jam})\quad \frac{\begin{array}{c} \Gamma \vdash FST_d\ MST_d \diamond_{\text{ModuleType}} \\ \Gamma \vdash FST_i\ MST_i \diamond_{\text{ModuleType}} \end{array}}{\begin{array}{l} \Gamma \vdash\ FST_d\ MST_d \\ \qquad \texttt{inherited}\ FST_i\ MST_i \diamond_{\text{MixinType}} \end{array}} \quad MST_i[MST_d]_\Gamma \neq \bot$$

Fig. 16. Well-formed types II.

Definition 3.1. For each pair FST, FST' of valid fields types, we denote by $FST\ [FST']$ the fields type uniquely defined by

$$FST\ [FST'](f) = \begin{cases} FST'(f) & \text{if } f \in Dom(FST'), \\ FST(f) & \text{otherwise.} \end{cases}$$

Analogously, for each pair MST, MST' of valid methods types and Γ environment, if the following condition holds:

$$\left.\begin{array}{l} msig \in Dom(MST) \cap Dom(MST') \\ MST(msig) = K\ RT\ \texttt{throws}\ ET \\ MST'(msig) = K'\ RT'\ \texttt{throws}\ ET' \end{array}\right\} \Rightarrow \left\{\begin{array}{l} K' = \texttt{static} \Leftrightarrow K = \texttt{static}, \\ \Gamma \vdash ET' \leq_e ET, \\ RT' = RT. \end{array}\right.$$

then, we denote by $MST\ [MST']_\Gamma$ the (valid) methods type uniquely defined by

$$MST\ [MST']_\Gamma(msig) = \begin{cases} MST'(msig) & \text{if } msig \in Dom(MST'), \\ MST(msig) & \text{otherwise;} \end{cases}$$

otherwise, $MST\ [MST']_\Gamma = \bot$.

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 (JLS 8.4.6.1, 8.4.6.2);
—a method overriding another cannot throw an exception which is not a subtype of some exception thrown by the overridden method (JLS 8.4.6.3);

$$(46)\quad \frac{\begin{array}{l}\Gamma \vdash I \text{ is}_i\ MST \quad \Gamma \vdash MST \diamond_{\text{InterfaceType}} \\ \Gamma \vdash I_1 : \emptyset\ MST_1 \ldots \Gamma \vdash I_n : \emptyset\ MST_n\end{array}}{\Gamma \vdash I : \emptyset\ (MST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} MST_n)[MST]_\Gamma} \quad \begin{array}{l} n \geq 0 \\ \{I_1, \ldots, I_n\} = \mathit{ParentInterfaces}(\Gamma, I)\end{array}$$

Set $MXT = FST_d\ MST_d\ \texttt{inherited}\ FST_i\ MST_i$

$$(47\text{-Jam})\quad \frac{\begin{array}{l}\Gamma \vdash M \text{ is}_m\ MXT \\ \Gamma \vdash MXT \diamond_{\text{MixinType}} \\ \Gamma \vdash I_1 : \emptyset\ MST_1 \ldots \Gamma \vdash I_n : \emptyset\ MST_n\end{array}}{\begin{array}{l}\Gamma \vdash M :\ FST_i[FST_d] \\ \qquad (MST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} MST_n)[MST_i[MST_d]_\Gamma]_\Gamma\end{array}} \quad \begin{array}{l} n \geq 0 \\ \{I_1, \ldots, I_n\} = \\ \qquad \mathit{DImplementedInterfaces}(\Gamma, M) \\ \neg \mathit{BadOverloading}(\Gamma, MST_i)\end{array}$$

$$(48)\quad \frac{}{\Gamma \vdash \texttt{Object} : \emptyset\ \emptyset}$$

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

$$(49\text{-[Jam]})\quad \frac{\begin{array}{l}\Gamma \vdash C \text{ is}_c\ CT \\ \Gamma \vdash CT \diamond_{\text{ClassType}} \\ \Gamma \vdash C <^1_c C' \\ \Gamma \vdash C' : FST'\ MST' \\ \Gamma \vdash I_1 : \emptyset\ MST_1 \ldots \Gamma \vdash I_n : \emptyset\ MST_n\end{array}}{\Gamma \vdash C : FST_c\ MST_c} \quad \begin{array}{l} n \geq 0 \\ \{I_1, \ldots, I_n\} = \\ \qquad \mathit{DImplementedInterfaces}(\Gamma, C) \\ \mathit{Kind}(MST_c) = \texttt{abstract} \Rightarrow K = \texttt{abstract} \\ [C \in \mathit{StandardClasses}(\Gamma)]\end{array}$$

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

$$(50\text{-Jam})\quad \frac{\begin{array}{l}\Gamma \vdash C \text{ is}_c\ CT \quad \Gamma \vdash CT \diamond_{\text{ClassType}} \\ \Gamma \vdash C \lhd_m M \\ \Gamma \vdash\ M \text{ is}_m\ FST_d\ MST_d\ \texttt{inherited} \\ \qquad FST_i\ MST_i \\ \Gamma \vdash M : FST_m\ MST_m \\ \Gamma \vdash C <^1_c C' \quad \Gamma \vdash C' : FST'\ MST' \\ \Gamma \vdash FST'\ MST' \leq_{mod} FST_i\ MST_i \\ \Gamma \vdash I_1 : \emptyset\ MST_1 \ldots \Gamma \vdash I_n : \emptyset\ MST_n\end{array}}{\Gamma \vdash C : FST_c\ MST_c} \quad \begin{array}{l} n \geq 0 \\ \{I_1, \ldots, I_n\} = \\ \qquad \mathit{DImplementedInterfaces}(\Gamma, M) \\ \mathit{Kind}(MST_c) = \texttt{abstract} \Rightarrow K = \texttt{abstract} \\ \neg \mathit{MayInterfere}(MST_m, MST' \setminus MST_i)\end{array}$$

Fig. 17. Type assignments.

—a method overriding another cannot have a different return type (JLS 8.4.6.3).

On interface types (that is, methods types where all methods are `abstract`) we define moreover a sum operation $\overset{\Gamma}{\oplus}$ which is basically set union, except that, in the case of two methods with the same signature, the operation is defined only if they have the same return type. In this case, the sum contains 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 parent class and implemented interfaces) many `abstract` methods which differ only in the `throws` clause (JLS 8.4.6.4). Note that in this case, differently to what happens in the case of overriding (modeled by the update operation on methods types), no check is required on `throws` clauses.

Definition 3.2. For each pair ET_1, ET_2 of exceptions types and Γ environment, we denote by $ET_1 \overset{\Gamma}{\otimes} ET_2$ the exceptions type defined by
$E \in ET_1 \overset{\Gamma}{\otimes} ET_2$ iff $\exists E_1 \in ET_1, E_2 \in ET_2$ such that $\Gamma \vdash E \leq_c E_1, \Gamma \vdash E \leq_c E_2$.

Note that, since in Java exceptions are classes and a class has only one parent,[11] the above condition can be equivalently expressed as follows:
$E \in ET_1 \overset{\Gamma}{\otimes} ET_2$ iff either $E \in ET_1$ and $\exists E_2 \in ET_2$ such that $\Gamma \vdash E \leq_c E_2$ or conversely.

LEMMA 3.3. *For each environment Γ, the operation $\overset{\Gamma}{\otimes}$ on exceptions types is commutative and associative.*

Definition 3.4. For each pair MST, MST' of valid methods type and Γ environment, if the following condition holds:

$$\left.\begin{array}{l} msig \in Dom(MST) \cap Dom(MST'), \\ MST(msig) = K\ RT\ \texttt{throws}\ ET, \\ MST'(msig) = K'\ RT'\ \texttt{throws}\ ET' \end{array}\right\} \Rightarrow \left\{\begin{array}{l} K' = K = \texttt{abstract}, \\ RT' = RT, \end{array}\right.$$

then, we denote by $MST \overset{\Gamma}{\oplus} MST'$ the (valid) methods type defined by

$$(MST \overset{\Gamma}{\oplus} MST')(msig) = \left\{\begin{array}{ll} MST\ (msig) & \text{if } msig \in Dom(MST), \\ & \quad msig \notin Dom(MST') \\ MST'(msig) & \text{if } msig \in Dom(MST'), \\ & \quad msig \notin Dom(MST) \\ \texttt{abstract}\ RT\ ET \overset{\Gamma}{\otimes} ET' & \text{if } MST(msig) = \\ & \quad \texttt{abstract}\ RT\ ET, \\ & \quad MST'(msig) = \\ & \quad \texttt{abstract}\ RT\ ET' \end{array}\right.$$

otherwise $MST \overset{\Gamma}{\oplus} MST' = \bot$.

LEMMA 3.5. *For each environment Γ, the operation $\overset{\Gamma}{\oplus}$ on methods type is commutative and associative.*

Associativity allows us to unambiguously write the sum of more than two methods types. Associativity and commutativity of the sum of methods type corresponds to the fact that the `implements` operator can be seen as a relation between a class and a set of interfaces, and analogously for the `extends` operator among interfaces.

The metarule (46) defines the type of an interface, which is a module type consisting of an empty fields type and a methods type. This methods type consists of the sum of the methods types of the superinterfaces, updated by the methods declared in the interface.

[11]Apart from `Object`, which is not an exception.

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

$$MayInterfere(T_1 \ldots T_n, T'_1 \ldots T'_k) = \; n = k \;\wedge \exists i \; s.t. \; T_i \neq T'_i \wedge \forall i \in \{1,\ldots,n\} \begin{cases} IsRef(T_i) \implies IsRef(T'_i), \\ IsPrim(T_i) \implies T_i = T'_i \end{cases}$$

$$\begin{aligned} MayInterfere(MST, MST') = \; & \exists msig : _ \in MST, msig' : _ \in MST' \; s.t. \\ & Name(msig) = Name(msig') \wedge \\ & (MayInterfere(Args(msig), Args(msig')) \\ & \vee (Args(msig) = Args(msig') \wedge \\ & Kind(msig), Kind(msig') \neq \texttt{static})) \end{aligned}$$

$$\begin{aligned} BadOverloading(\Gamma, MST) = \; & \exists msig : _, msig' : _ \in MST, msig \neq msig' \; s.t. \\ & Name(msig) = Name(msig'), \\ & Args(msig) = T_1 \ldots T_n, Args(msig') = T'_1 \ldots T'_n, \\ & \forall j \in \{1,\ldots,n\} \; \Gamma \vdash T_j \leq T'_j \end{aligned}$$

Fig. 18. Definition of *MayInterfere* and *BadOverloading*.

The metarule (47) defines the type of a mixin, which is a module type, that is, a pair consisting of a fields type 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 methods obtained updating inherited methods by the defined methods. Here and in metarules (49–50), this models the fact that if there is a nonabstract method in the superclass with the same signature of some methods of the superinterfaces, then the check that this method has a compatible `throws` clause must be performed only if it is inherited, that is, not overridden (JLS 8.4.6.4). The last side condition expresses the constraint on overloading among `inherited` methods informally described in Section 2.4: the predicate *BadOverloading*, defined in Figure 18, is true on a methods type *MST* whenever it contains a pair of methods with the same name and arguments type in the subtyping relation. The symbol _ is used as a wild card.

The three metarules (48–50) define the type of a class, which is a module type, that is, a pair consisting of a fields and a methods type.

(48) For simplicity, we have ignored all the predefined methods of `Object`, defined in JLS 4.3.2.

(49) 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 and the abstract methods of the superclass, updated by the methods obtained updating the nonabstract methods of the superclass updated by those declared in the class. The third side condition (here and in the next metarule) expresses the constraint that a class with `abstract` methods must be declared `abstract` (JLS 8.4.3.1). The last side condition applies only to Jam.

(50) 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

$$(51)\quad \overline{\Gamma \vdash \emptyset \diamond}$$

$$(52)\quad \frac{\Gamma \vdash \Gamma' \diamond \quad \Gamma \vdash C : FST\ MST}{\Gamma \vdash \Gamma' \cup\ \{C \text{ is}_c\ CT, C <^1_c C', C \lhd^1_i I_1, \ldots, C \lhd^1_i I_n\} \diamond} \quad \Gamma'(C) = \bot$$

$$(53\text{-Jam})\quad \frac{\Gamma \vdash \Gamma' \diamond \quad \Gamma \vdash C : FST\ MST}{\Gamma \vdash\ \Gamma' \cup \{C \text{ is}_c\ K\ KST\ \emptyset\ \emptyset, C <^1_c C', C \lhd_m M, C \lhd^1_i I_1, \ldots, C \lhd^1_i I_n\} \diamond} \quad \Gamma'(C) = \bot \quad MST$$

$$(54)\quad \frac{\Gamma \vdash \Gamma' \diamond \quad \Gamma \vdash I : \emptyset\ MST}{\Gamma \vdash\ \Gamma' \cup \{I \text{ is}_i\ IT, I <^1_i I_1, \ldots, I <^1_i I_n\} \diamond} \quad \Gamma'(I) = \bot$$

$$(55\text{-Jam})\quad \frac{\Gamma \vdash \Gamma' \diamond \quad \Gamma \vdash M : FST\ MST}{\Gamma \vdash \Gamma' \cup \{M \text{ is}_m\ MXT, M \lhd^1_i I_1, \ldots, M \lhd^1_i I_n\} \diamond} \quad \Gamma'(M) = \bot$$

Fig. 19. Well-formed class, interface, and mixin declarations.

of the sum of the methods types of the implemented interfaces and the abstract methods of the superclass, updated by methods obtained updating the nonabstract methods of the superclass by those defined in the mixin. The fourth side condition expresses the constraint on instantiation related to interferences between an extra-method in the parent and a method in the mixin informally described in Section 2.3: the predicate *MayInterfere*, defined in Figure 18, is true on two methods types *MST*, *MST′* whenever there is a pair of methods, the former in *MST* and the latter in *MST′*, with the same name and interfering arguments type.

Recall that, in case of multiple instantiations of a mixin type which declares fields, the semantics of field accesses through a reference of the mixin type is nonobvious (see the discussion in Section 2.2.6). This situation could be avoided by adding the following side condition:

$$\Gamma \vdash C' \leq M \Rightarrow FST_d = \emptyset, FST_i = \emptyset.$$

However, the Jam compiler only produces a warning in this case.

We assume that the metarules in Figure 17 can be instantiated only when the sum and update operations are defined.

3.5.5 *Well-Formedness of Environments.* The metarules in Figure 19 express the fact that 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 Drossopoulou and Eisenbach [1999] of considering a 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 $\Gamma \vdash \diamond$.

The four metarules (52–55) all express the fact that the declaration of a Jam module T (standard Java heir class, mixin instance, interface, and mixin, respectively) is well-formed with respect to an environment Γ if T can be correctly typed in Γ and T has not been previously declared (side condition). Note

type-assertion	::=	*modname* $:^{+}$ *annotated-meths-type* |
		params $\diamond_{\text{Params}}$ |
		body-decl $\diamond_{\text{Body-Decl}}$ |
		body-decls $\diamond_{\text{Body-Decls}}$
judgment	::=	*env*, *modname* $\vdash$ *meth* $\diamond_{\text{Meth}}$ |
		env, *modname* $\vdash$ *constructor* $\diamond_{\text{Constr}}$ |
		local-env, *env*, *modname*, *kind* $\vdash$ *expr* : $\langle$*type*, *exc-type*$\rangle$ |
		local-env, *env*, *modname*, *kind* $\vdash$ *stmt* : *exc-type* |
		local-env, *env*, *modname*, *kind* $\vdash$ *stmts* : *exc-type* |
		local-env, *env*, *modname*, *kind* $\vdash$ *mbody* : $\langle$*type*, *exc-type*$\rangle$
local-env	::=	$\langle$`name` *simple-type*$\rangle^{\circledast}$
kind	::=	`instance` | `static` | `constructor`

Fig. 20. Judgments II.

that T cannot be correctly typed if there is a cycle in the inheritance hierarchy involving T.

3.6 Typing Rules II

In this subsection, we give the second part of the metarules of the Jam type system, that is, those related to body declarations. The syntax of the judgments which appear in these metarules is given in Figure 20. Besides judgments of the form $\Gamma \vdash \gamma$ as those introducedin the preceding subsection, we also consider here other two types of judgments:

—$\Gamma, T \vdash \gamma$,
—$\Pi, \Gamma, T, K \vdash \gamma$,

where Π is a *local environment*, T is a module (either class, or interface, or mixin) name, and K is a (body) kind (either `instance` or `static` or `constructor`). The local environment contains type assignments for formal parameters and variables introduced in `catch` clauses (for simplicity we do not consider other local variables). We use in the following Π(`id`) to indicate the type associated to the identifier `id`. The type assertions in Figure 20 have the following informal meaning:

—$T{:}^{+}AMST$: the module (either class, interface, or mixin) T has the annotated methods type (see below) $AMST$.
—$P \diamond_{\text{Params}}$: P are well-formed parameters.
—$BD\diamond_{\text{Body-Decl}}$: BD is a well-formed body declaration.
—$\{BD_1, \ldots, BD_n\}\diamond_{\text{Body-Decls}}$: $BD_1, \ldots, BD_n$ are well-formed body declarations.

The judgments in Figure 20 have the following informal meaning.

—$\Gamma, T \vdash M \diamond_{\text{Meth}}$: M is a well-formed method in either a class or mixin T.
—$\Gamma, C \vdash CN\diamond_{\text{Constr}}$: CN is a well-formed constructor in a class C.
—$\Pi, \Gamma, T, K \vdash E$: $\langle T', ET\rangle$: E is a well-formed expression of type T' and can throw the exceptions ET in an either class or mixin T, in a method body of kind K with respect to the local environment Π and the global environment Γ.

$$\textit{annotated-meths-type} ::= \langle \textit{modname meth-sig} : \textit{meth-type} \rangle^{\circledast}$$

Fig. 21. Syntax of annotated methods type.

$$\frac{\Gamma \vdash MT_1 \diamond_{\text{MethType}} \cdots \quad \Gamma \vdash MT_n \diamond_{\text{MethType}}}{\Gamma \vdash \{T_1\ msig_1 : MT_1, \ldots, T_n\ msig_n : MT_n\} \diamond_{\text{AnnotatedMethsType}}} \quad \begin{array}{l} n \geq 0 \\ i \neq j \Rightarrow T_i\ msig_i \neq T_j\ msig_j \\ i \neq j \wedge msig_i = msig_j \Rightarrow \\ \quad \begin{cases} Kind(MT_i) = Kind(MT_j) = \texttt{abstract} \\ Ret(MT_i) = Ret(MT_j) \end{cases} \end{array}$$

Fig. 22. Well-formed annotated methods type.

— $\Pi, \Gamma, T, K \vdash S: ET: S$ is a well-formed statement that can throw exceptions ET in an either class or mixin T, in a method body of kind K with respect to the local environment Π and the global environment Γ.

— $\Pi, \Gamma, T, K \vdash SS: ET: SS$ are well-formed statements that can throw exceptions ET in an either class or mixin T, in a method body of kind K with respect to the local environment Π and the global environment Γ.

— $\Pi, \Gamma, T, K \vdash MB: \langle T', ET \rangle: MB$ is a well-formed method body of type T' that can throw exceptions ET in an either class or mixin T, in a method body of kind K with respect to the local environment Π and the global environment Γ.

In order to formally define the well-formedness of body declarations, it is necessary to introduce *annotated methods types*, whose syntax is given in Figure 21. Indeed, for solving overloading in method calls (JLS 15.11.2.2) it is necessary to know, for each method, the module where it has been declared.

An annotated methods type is valid whenever it does not contain two methods with the same signature and annotation and, moreover, if there are more methods with the same signature they must be abstract and have the same return type. An annotated methods type is well-formed with respect to an environment Γ if it is valid, as shown in Figure 22.

Module names used in annotations are either class, or interface, or mixin names or *Parent* names which represent the generic parent of a mixin; see Section 2.4. Formally, we add the production in Figure 23 to the definition of Jam module names (Figure 10), and the metarule in Figure 24 to those defining the widening relation (Figure 13).

We define moreover:

$$\begin{aligned} &Annotate(T, \{msig_1 : MT_1, \ldots, msig_n : MT_n\}) \\ &\quad = \{T\ msig_1 : MT_1, \ldots, T\ msig_n : MT_n\}. \end{aligned}$$

In the following we will consider only valid annotated methods types and use the following notations: if $AMST = \{T_1\ msig_1 : MT_1, \ldots, T_n\ msig_n : MT_n\}$ is a valid annotated methods type, then $Dom(AMST) = \{T_1\ msig_1, \ldots, T_n\ msig_n\}$ and $AMST(T\ msig) = MT_i$ if $T\ msig = T_i\ msig_i$ for some $i \in \{1, \ldots, n\}$, $\perp$ otherwise. The definitions of the *Abstract* and *NonAbstract* functions on annotated methods types are analogous to those on methods types. The update and sum operations on annotated methods types are defined as follows.

$modname ::= Parent(\texttt{mxname})$

Fig. 23. *Parent* names.

$$\text{(56-Jam)}\quad \frac{\Gamma \vdash M \text{ is}_m\ MXT}{\begin{array}{l}\Gamma \vdash M \leq Parent(M)\\ \Gamma \vdash Parent(M) \leq Parent(M)\end{array}}$$

Fig. 24. Extension of the *widening* relation.

Definition 3.6. For each pair $AMST, AMST'$ of valid annotated methods types and Γ environment, if the following condition holds:

$$\left.\begin{array}{l} T\ msig \in Dom(AMST),\\ T'\ msig \in Dom(AMST'),\\ AMST\,(T\ msig) = K\ RT\ \texttt{throws}\ ET,\\ AMST'(T'\ msig) = K'\ RT'\ \texttt{throws}\ ET', \end{array}\right\} \Rightarrow \left\{\begin{array}{l} K' = \texttt{static} \Leftrightarrow K = \texttt{static},\\ \Gamma \vdash ET' \leq_e ET,\\ RT' = RT, \end{array}\right.$$

then, we denote by $AMST\ [AMST']_\Gamma$ the (valid) annotated methods type uniquely defined by

$$AMST\,[AMST']_\Gamma(T\ msig) = \begin{cases} \bot & \text{if } \exists T' \neq T \text{ such that } (T'\ msig) \\ & \quad \in Dom(AMST'),\\ AMST'(T\ msig) & \text{if } T\ msig \in Dom(AMST'),\\ AMST\,(T\ msig) & \text{otherwise;} \end{cases}$$

otherwise, $AMST\ [AMST']_\Gamma = \bot$.

In the case of annotated method type the sum operation $\overset{\Gamma}{\oplus}$ is actually set union, except for being partially defined.

Definition 3.7. For each pair $AMST, AMST'$ of valid annotated methods types and Γ environment, if the following condition holds:

$$\left.\begin{array}{l} T\ msig \in Dom(AMST),\\ T'\ msig \in Dom(AMST'),\\ AMST\,(T\ msig) = K\ RT\ \texttt{throws}\ ET,\\ AMST'(T'\ msig) = K'\ RT'\ \texttt{throws}\ ET', \end{array}\right\} \Rightarrow \left\{\begin{array}{l} K' = K = \texttt{abstract},\\ RT' = RT, \end{array}\right.$$

then we denote by $AMST \overset{\Gamma}{\oplus} AMST'$ the (valid) annotated methods type defined by $AMST \overset{\Gamma}{\oplus} AMST' = AMST \cup AMST'$; otherwise $AMST \overset{\Gamma}{\oplus} AMST' = \bot$.

Figure 25 shows the metarules defining the annotated methods types of modules (the definition is analogous to that of methods types, except for the type checks that would be redundant).

We briefly illustrate now the metarules related to body declarations, given in bottom-up order.

(57)
$$\frac{\Gamma \vdash I \text{ is}_i\ MST \quad \Gamma \vdash I_1 :^+ AMST_1 \ldots \Gamma \vdash I_n :^+ AMST_n}{\Gamma \vdash I :^+ (AMST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} AMST_n)[Annotate(I, MST)]_\Gamma} \quad \begin{array}{l} n \geq 0 \\ \{I_1, \ldots, I_n\} = ParentInterfaces(\Gamma, I) \end{array}$$

(58-Jam)
$$\frac{\Gamma \vdash M \text{ is}_m\ FST_d\ MST_d\ \texttt{inherited}\ FST_i\ MST_i \quad \Gamma \vdash I_1 :^+ AMST_1 \ldots \Gamma \vdash I_n :^+ AMST_n}{\begin{array}{l} \Gamma \vdash M :^+ (AMST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} AMST_n)[Annotate(Parent(M), MST_i)[Annotate(M, MST_d)]_\Gamma]_\Gamma \\ \Gamma \vdash Parent(M) :^+ Annotate(Parent(M), MST_i) \end{array}} \quad \begin{array}{l} n \geq 0 \\ \{I_1, \ldots, I_n\} = \\ DImplementedInterfaces(\Gamma, M) \end{array}$$

(59)
$$\frac{}{\Gamma \vdash \texttt{Object} :^+ \emptyset}$$

(60-[Jam])
$$\frac{\Gamma \vdash C \text{ is}_c\ _\ _\ _\ MST \quad \Gamma \vdash C <^1_c C' \quad \Gamma \vdash C' :^+ AMST' \quad \Gamma \vdash I_1 :^+ AMST_1 \ldots \Gamma \vdash I_n :^+ AMST_n}{\Gamma \vdash C :^+ (AMST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} AMST_n \overset{\Gamma}{\oplus} Abstract(AMST'))[NonAbstract(AMST')[Annotate(C, MST)]_\Gamma]_\Gamma} \quad \begin{array}{l} n \geq 0 \\ [C \in StandardClasses(\Gamma)] \\ \{I_1, \ldots, I_n\} = \\ DImplementedInterfaces(\Gamma, C) \end{array}$$

(61-Jam)
$$\frac{\Gamma \vdash C \lhd_m M \quad \Gamma \vdash M \text{ is}_m\ FST_d\ MST_d\ \texttt{inherited}\ FST_i\ MST_i \quad \Gamma \vdash I_1 :^+ AMST_1 \ldots \Gamma \vdash I_n :^+ AMST_n \quad \Gamma \vdash C <^1_c C' \quad \Gamma \vdash C' :^+ AMST'}{\Gamma \vdash C :^+ (AMST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} AMST_n \overset{\Gamma}{\oplus} Abstract(AMST'))[NonAbstract(AMST')[Annotate(C, MST_d)]_\Gamma]_\Gamma} \quad \begin{array}{l} \{I_1, \ldots, I_n\} = \\ DImplementedInterfaces(\Gamma, M) \end{array}$$

Fig. 25. Annotated methods types for interfaces, mixin and classes.

(62)
$$\frac{}{\begin{array}{l}\Pi, \Gamma, T, K \vdash \texttt{true} : \langle \texttt{boolean}, \emptyset \rangle \\ \Pi, \Gamma, T, K \vdash \texttt{false} : \langle \texttt{boolean}, \emptyset \rangle\end{array}}$$

(63)
$$\frac{}{\Pi, \Gamma, T, K \vdash \texttt{null} : \langle \texttt{nil}, \emptyset \rangle}$$

(64)
$$\frac{}{\Pi, \Gamma, T, K \vdash n : \langle \texttt{int}, \emptyset \rangle}$$

Fig. 26. Literals.

Literals are always well-formed and their evaluation throws no exceptions ((62–64) in Figure 26).

Local variable identifiers are well-formed if defined in the local environment, and their evaluation throws no exceptions ((65) in Figure 27).

The expression `this` can be used only in instance methods and constructors ((66) in Figure 27).

The class instance creation expression is well-formed ((67) in Figure 27) if there exists the most specific among the applicable constructors (JLS 15.9); the definition of *MostSpec* for constructors is shown in Figure 28. The exceptions possibly thrown by the creation expression are those declared in the selected constructor, plus all those possibly thrown by the arguments. In the side condition, the restriction $E_j \neq \texttt{this}$ (that is, the keyword `this` cannot be used as an

$$(65)\quad \frac{}{\Pi, \Gamma, T, K \vdash \texttt{id} : \langle \Pi(\texttt{id}), \emptyset \rangle} \quad \Pi(\texttt{id}) \neq \bot$$

$$(66)\quad \frac{}{\Pi, \Gamma, T, K \vdash \texttt{this} : \langle T, \emptyset \rangle} \quad K \in \{\texttt{instance}, \texttt{constructor}\}$$

$$(67\text{-[Jam]})\quad \frac{\begin{array}{c}\Gamma \vdash C \text{ is}_c \texttt{ concrete } \{KT_1, \ldots, KT_n\} \text{ - -} \\ \forall j \in \{1, \ldots, m\}\ \Pi, \Gamma, T, K \vdash E_j : \langle T_j, ET_j \rangle\end{array}}{\begin{array}{l}\Pi, \Gamma, T, K \vdash \texttt{ new } C(E_1, \ldots, E_m) \\ \quad : \langle C, Exc(KT) \cup \bigcup_{j \in \{1,\ldots,m\}} ET_j \rangle\end{array}} \quad \begin{array}{l} n \geq 1 \\ m \geq 0 \\ KT = MostSpec(\Gamma, \\ \quad \{KT_1, \ldots, KT_n\}, \\ \quad T_1 \ldots T_m) \\ KT \neq \bot \\ [T \in Mixins(\Gamma) \Rightarrow \forall j\ E_j \neq \texttt{this}] \end{array}$$

Fig. 27. Local variables, `this`, and instance creation expressions.

$$MostSpec(\Gamma, \{KT_1, \ldots, KT_n\}, AT) = \begin{cases} KT & \text{if } \exists KT \in A = ApplConstrs(\Gamma, \{KT_1, \ldots, KT_n\}, AT) \textit{ s.t.} \\ & \quad \forall KT' \in A,\ MoreSpec(\Gamma, KT, KT') \\ \bot & \text{otherwise} \end{cases}$$

$$MoreSpec(\Gamma, KT, KT') = \begin{cases} Args(KT) = T_1 \ldots T_n \\ Args(KT') = T'_1 \ldots T'_n \\ \forall j \in \{1, \ldots, n\}\ \Gamma \vdash T_j \leq T'_j \end{cases}$$

$$ApplConstrs(\Gamma, \{KT_1, \ldots, KT_n\}, T_1 \ldots T_n) = \{KT_i | Args(KT_i) = T'_1 \ldots T'_n \wedge \forall j \in \{1, \ldots, n\}\ \Gamma \vdash T_j \leq T'_j\}$$

Fig. 28. Definition of *MostSpec* for constructors (*s.t.* = such that).

argument) is due to the problems related to overloading resolution explained in Section 2.5. The metarules from (68) to (72) (Figure 29) are related to (instance, static and via `super`) field selection.

In method invocation (metarules from (73) to (77); see Figure 31), the Java rule for overloading resolution requires to find the most specific method among those applicable (JLS 15.12); Figure 30 shows the definition of *MostSpec* for methods. The expression `this` cannot be used in arguments of method invocation, as in metarule (67).

In an assignment (78) (see Figure 32), the type of the right-hand expression must be a subtype of that of the variable, as it usually happens in object-oriented languages and in particular in Java (JLS 15.26.1). In Jam the only difference is that we have also mixin types among types.

The block `try`/`catch`/`finally` is the only statement which removes some checked exceptions (those captured by the `catch` clauses). In metarule (83) (see Figure 33), $\overset{\Gamma}{\ominus}$ denotes the *difference* of exceptions types, defined below.

Definition 3.8. For each pair ET, ET' of exceptions types and Γ environment, we denote by $ET \overset{\Gamma}{\ominus} ET'$ the exceptions type defined by

$$E \in ET \overset{\Gamma}{\ominus} ET' \text{ iff } E \in ET \text{ and } \nexists E' \in ET' \text{ such that } \Gamma \vdash E \ \leq_c \ E'.$$

The metarules in Figures 34 and 36 are self-explanatory. Metarule (89) in Figure 35 defines well-formed constructor declarations: recall from Section 3.2 that we do not consider the invocation via `this` of another constructor of the same class since this is just a syntactic abbreviation.

$$(68)\quad \frac{\begin{array}{l}\Pi, \Gamma, T, K \vdash E : \langle T', ET\rangle \\ \Gamma \vdash T' : \{f_1 : FT_1, \ldots, f_n : FT_n\}\ _\end{array}}{\Pi, \Gamma, T, K \vdash E.f_i : \langle \mathit{Type}(FT_i), ET\rangle} \quad 1 \leq i \leq n$$

$$(69)\quad \frac{\Gamma \vdash T : \{f_1 : FT_1, \ldots, f_n : FT_n\}\ _}{\Pi, \Gamma, T, K \vdash f_i : \langle \mathit{Type}(FT_i), \emptyset\rangle} \quad \begin{array}{l} 1 \leq i \leq n \\ \Pi(f_i) = \bot \\ K = \texttt{static} \Rightarrow \mathit{Kind}(FT_i) = \texttt{static}\end{array}$$

$$(70)\quad \frac{\Gamma \vdash C : \{f_1 : FT_1, \ldots, f_n : FT_n\}\ _}{\Pi, \Gamma, T, K \vdash C.f_i : \langle \mathit{Type}(FT_i), \emptyset\rangle} \quad \begin{array}{l} 1 \leq i \leq n \\ C \in \mathit{Classes}(\Gamma) \\ \Pi(C) = \bot \\ \mathit{Kind}(FT_i) = \texttt{static}\end{array}$$

$$(71)\quad \frac{\Gamma \vdash C <^1_c C' \quad \Gamma \vdash C' : \{f_1 : FT_1, \ldots, f_n : FT_n\}_}{\Pi, \Gamma, C, \texttt{instance} \vdash \texttt{super}.f_i : \langle \mathit{Type}(FT_i), \emptyset\rangle} \quad 1 \leq i \leq n$$

$$(72)\quad \frac{\Gamma \vdash M \text{ is}_m\ _\ _\ \texttt{inherited}\ \{f_1 : FT_1, \ldots, f_n : FT_n\}\ _}{\Pi, \Gamma, M, \texttt{instance} \vdash \texttt{super}.f_i : \langle \mathit{Type}(FT_i), \emptyset\rangle} \quad 1 \leq i \leq n$$

Fig. 29. Field accesses.

$$\mathit{MostSpec}(\Gamma, AMST, m, AT) = \begin{cases} MT & \text{if } \exists T\ msig : MT \in A = \mathit{ApplMeths}(\Gamma, AMST, m, AT)\ s.t. \\ & \quad \forall T'\ msig' : MT' \in A,\ \mathit{MoreSpec}(\Gamma, T\ msig : MT, T'\ msig' : MT') \\ \bot & \text{otherwise}\end{cases}$$

$$\mathit{MoreSpec}(\Gamma, T\ msig : MT, T'\ msig' : MT') = \begin{cases} \Gamma \vdash T \leq T' \\ \mathit{Args}(msig) = T_1 \ldots T_n \\ \mathit{Args}(msig') = T'_1 \ldots T'_n \\ \forall j \in \{1, \ldots, n\}\ \Gamma \vdash T_j \leq T'_j \end{cases}$$

$$\mathit{ApplMeths}(\Gamma, \{\widehat{T_1}\ msig_1 : MT_1, \ldots, \widehat{T_k}\ msig_k : MT_k\}, m, T_1 \ldots T_n) =$$
$$\{\widehat{T_i}\ msig_i : MT_i | \mathit{Name}(msig_i) = m, \mathit{Args}(msig_i) = T'_1 \ldots T'_n \wedge \forall j \in \{1, \ldots, n\}\ \Gamma \vdash T_j \leq T'_j\}$$

Fig. 30. Definition of *MostSpec* for methods (*s.t.* = such that).

4. JAM TO JAVA TRANSLATION

In this section, we give a formal definition of the dynamic semantics of Jam directly by giving a translation in Java. The same approach of defining a Java extension by translation into Java as been taken for Pizza [Odersky and Wadler 1997], a superset of Java which incorporates parametric polymorphism, higher-order functions and algebraic data types, and its evolution GJ (for "Generic Java") [Bracha et al. 1998].

We first illustrate informally the basic ideas through some examples (Section 4.1), then provide the formal definition (Section 4.2); finally in Section 4.3 we prove 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 Section 2. Hence, the two

(73-[Jam])
$$\frac{\Pi,\Gamma,T,K\vdash E:\langle T',ET\rangle \quad \Gamma\vdash T' :^{+} AMST \quad \forall i\in\{1,\dots,n\}\ \Pi,\Gamma,T,K\vdash E_i:\langle T_i,ET_i\rangle}{\Pi,\Gamma,T,K\vdash E.m(E_1,\dots,E_n):\langle Ret(MT), Exc(MT)\cup ET \cup\bigcup_{i\in\{1,\dots,n\}} ET_i\rangle}$$
$n\geq 0$, $MT = MostSpec(\Gamma, AMST, m, T_1\dots T_n)$, $[T\in Mixins(\Gamma)\Rightarrow \forall j\ E_j\neq \texttt{this}]$

(74-[Jam])
$$\frac{\Gamma\vdash T :^{+} AMST \quad \forall i\in\{1,\dots,n\}\ \Pi,\Gamma,T,K\vdash E_i:\langle T_i,ET_i\rangle}{\Pi,\Gamma,T,K\vdash m(E_1,\dots,E_n):\langle Ret(MT), Exc(MT)\cup\bigcup_{i\in\{1,\dots,n\}} ET_i\rangle}$$
$n\geq 0$, $MT = MostSpec(\Gamma, AMST, m, T_1\dots T_n)$, $[T\in Mixins(\Gamma)\Rightarrow \forall j\ E_j\neq \texttt{this}]$, $K=\texttt{static}\Rightarrow Kind(MT)=\texttt{static}$

(75-[Jam])
$$\frac{\Gamma\vdash C :^{+} AMST \quad \forall i\in\{1,\dots,n\}\ \Pi,\Gamma,T,K\vdash E_i:\langle T_i,ET_i\rangle}{\Pi,\Gamma,T,K\vdash C.m(E_1,\dots,E_n):\langle Ret(MT), Exc(MT)\cup\bigcup_{i\in\{1,\dots,n\}} ET_i\rangle}$$
$n\geq 0$, $MT = MostSpec(\Gamma, AMST, m, T_1\dots T_n)$, $C\in Classes(\Gamma)$, $\Pi(C)=\bot$, $Kind(MT)=\texttt{static}$, $[T\in Mixins(\Gamma)\Rightarrow \forall j\ E_j\neq \texttt{this}]$

(76)
$$\frac{\Gamma\vdash C <^1_c C' \quad \Gamma\vdash C' :^{+} AMST \quad \forall i\in\{1,\dots,n\}\ \Pi,\Gamma,C,K\vdash E_i:\langle T_i,ET_i\rangle}{\Pi,\Gamma,C,K\vdash \texttt{super}.m(E_1,\dots,E_n):\langle Ret(MT), Exc(MT)\cup\bigcup_{i\in\{1,\dots,n\}} ET_i\rangle}$$
$n\geq 0$, $K\neq\texttt{static}$, $MT = MostSpec(\Gamma, AMST, m, T_1\dots T_n)$, $Kind(MT)=\texttt{instance}$

(77-Jam)
$$\frac{\Gamma\vdash Parent(M) :^{+} AMST \quad \forall i\in\{1,\dots,n\}\ \Pi,\Gamma,M,\texttt{instance}\vdash E_i:\langle T_i,ET_i\rangle}{\Pi,\Gamma,M,\texttt{instance}\vdash \texttt{super}.m(E_1,\dots,E_n):\langle Ret(MT), Exc(MT)\cup\bigcup_{i\in\{1,\dots,n\}} ET_i\rangle}$$
$n\geq 0$, $MT = MostSpec(\Gamma, AMST, m, T_1\dots T_n)$, $Kind(MT)=\texttt{instance}$, $\forall j\ E_j\neq\texttt{this}$

Fig. 31. Method invocations.

(78)
$$\frac{\Pi,\Gamma,T,K\vdash E_1:\langle T_1,ET_1\rangle \quad \Pi,\Gamma,T,K\vdash E_2:\langle T_2,ET_2\rangle \quad \Gamma\vdash T_2\leq T_1}{\Pi,\Gamma,T,K\vdash E_1=E_2:\langle T_1, ET_1\cup ET_2\rangle}$$

Fig. 32. Assignments.

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 mixin (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.

$$(79)\quad \frac{\begin{array}{l}\Pi,\Gamma,T,K \vdash E : \langle \texttt{boolean}, ET\rangle \\ \Pi,\Gamma,T,K \vdash S_1 : ET_1 \\ \Pi,\Gamma,T,K \vdash S_2 : ET_2\end{array}}{\begin{array}{l}\Pi,\Gamma,T,K \vdash \texttt{if}\ (E)\ S_1\ \texttt{else}\ S_2 \\ \qquad : ET \cup ET_1 \cup ET_2\end{array}}$$

$$(80)\quad \frac{\begin{array}{l}\Pi,\Gamma,T,K \vdash S_1 : ET_1 \\ \ldots \\ \Pi,\Gamma,T,K \vdash S_n : ET_n\end{array}}{\Pi,\Gamma,T,K \vdash S_1 \ldots S_n : \bigcup_{j\in\{1,\ldots,n\}} ET_j} \quad n \geq 0$$

$$(81)\quad \frac{\Pi,\Gamma,T,K \vdash SS : ET}{\Pi,\Gamma,T,K \vdash \{SS\} : ET}$$

$$(82)\quad \frac{\begin{array}{l}\Pi,\Gamma,T,K \vdash E : \langle T', ET'\rangle \\ \Gamma \vdash T' \leq_c \texttt{Throwable}\end{array}}{\Pi,\Gamma,T,K \vdash \texttt{throw}\ E; : T' \cup ET'}$$

$$(83)\quad \frac{\begin{array}{l}\Pi,\Gamma,T,K \vdash SS : ET \\ \Pi[C_1/\texttt{id}_1],\Gamma,T,K \vdash SS_1 : ET_1,\ \Gamma \vdash C_1 \leq_c \texttt{Throwable} \\ \ldots \\ \Pi[C_n/\texttt{id}_n],\Gamma,T,K \vdash SS_n : ET_n,\ \Gamma \vdash C_n \leq_c \texttt{Throwable} \\ \Pi,\Gamma,T,K \vdash SS' : ET'\end{array}}{\begin{array}{l}\Pi,\Gamma,T,K \vdash \texttt{try}\ \{\ SS\ \} \\ \qquad \texttt{catch}\ C_1\ \texttt{id}_1\ \{\ SS_1\ \} \\ \qquad \ldots \\ \qquad \texttt{catch}\ C_n\ \texttt{id}_n\ \{\ SS_n\ \} \\ \qquad \texttt{finally}\ \{\ SS'\ \} \\ \qquad : \left(ET \overset{\Gamma}{\ominus} \bigcup_{i\in\{1,\ldots,n\}}\{C_i\}\right) \cup \left(\bigcup_{i\in\{1,\ldots,n\}} ET_i\right) \cup ET'\end{array}} \quad \begin{array}{l}\mathrm{Dom}(\Pi)\cap \\ \{\texttt{id}_1,\ldots, \\ \texttt{id}_n\} = \emptyset\end{array}$$

$$(84)\quad \frac{\Pi,\Gamma,T,K \vdash E : \langle T, ET\rangle}{\Pi,\Gamma,T,K \vdash E\ ; : ET}$$

$$(85)\quad \frac{\Pi,\Gamma,T,K \vdash SS : ET}{\Pi,\Gamma,T,K \vdash \{SS\ \texttt{return}\ ;\} : \langle \texttt{void}, ET\rangle}$$

$$(86)\quad \frac{\begin{array}{l}\Pi,\Gamma,T,K \vdash SS : ET \\ \Pi,\Gamma,T,K \vdash E : \langle T', ET'\rangle\end{array}}{\Pi,\Gamma,T,K \vdash \{SS\ \texttt{return}\ E\ ;\} : \langle T', ET \cup ET'\rangle}$$

Fig. 33. Statements and method bodies.

$$(87)\quad \frac{\Gamma \vdash ST_1 \diamond_{\text{SimpleType}} \ldots \Gamma \vdash ST_n \diamond_{\text{SimpleType}}}{\Gamma \vdash (ST_1\ \texttt{id}_1,\ldots,ST_n\ \texttt{id}_n)\diamond_{\text{Params}}} \quad \begin{array}{l}\forall i,j \in \{1,\ldots,n\} \\ i \neq j \Rightarrow \texttt{id}_i \neq \texttt{id}_j\end{array}$$

Fig. 34. Parameters.

The former 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).

$$(88)\quad \frac{\begin{array}{l}\Gamma \vdash RT\diamond_{\text{RetType}}\\ \Gamma \vdash (ST_1\ \mathtt{id}_1, \ldots, ST_n\ \mathtt{id}_n)\diamond_{\text{Params}}\\ \Gamma \vdash ET\diamond_{\text{ExcType}}\\ \{\mathtt{id}_1 \mapsto ST_1, \ldots, \mathtt{id}_n \mapsto ST_n\}, \Gamma, T, K' \vdash MB : \langle RT', ET'\rangle\\ \Gamma \vdash ET' \leq_e ET\\ \Gamma \vdash RT' \leq RT\end{array}}{\Gamma, T \vdash K\ RT\ \mathtt{id}\ (ST_1\ \mathtt{id}_1, \ldots, ST_n\ \mathtt{id}_n)\ \mathtt{throws}\ ET\ MB\diamond_{\text{Meth}}} \quad K' = \begin{cases}\mathtt{instance} & \text{if } K = \epsilon\\ K & \text{otherwise}\end{cases}$$

$$(89)\quad \frac{\begin{array}{l}\Gamma \vdash (ST_1\ \mathtt{id}_1, \ldots, ST_m\ \mathtt{id}_m)\diamond_{\text{Params}}\\ \Gamma \vdash ET\diamond_{\text{ExcType}}\\ \{\mathtt{id}_1 \mapsto ST_1, \ldots, \mathtt{id}_m \mapsto ST_m\},\\ \qquad \Gamma, C, \mathtt{constructor} \vdash E_1 : \langle T_1', ET_1'\rangle\\ \ldots\\ \{\mathtt{id}_1 \mapsto ST_1, \ldots, \mathtt{id}_m \mapsto ST_m\},\\ \qquad \Gamma, C, \mathtt{constructor} \vdash E_n : \langle T_n', ET_n'\rangle\\ \{\mathtt{id}_1 \mapsto ST_1, \ldots, \mathtt{id}_m \mapsto ST_m\},\\ \qquad \Gamma, C, \mathtt{constructor} \vdash SS : ET''\\ \Gamma \vdash \left(\bigcup_{i\in\{1,\ldots,n\}} ET_i'\right) \cup ET'' \cup \mathit{Exc}(KT) \leq_e ET\\ \Gamma \vdash C <_c^1 C'\\ \Gamma \vdash C'\ \text{is}_c\ _\ KST\ _\ _\end{array}}{\begin{array}{l}\Gamma, C \vdash\ C\ (ST_1\ \mathtt{id}_1, \ldots, ST_m\ \mathtt{id}_m)\ \mathtt{throws}\ ET\ \{\\ \qquad \mathtt{super}(E_1, \ldots, E_n)\ ;\ SS\ \}\diamond_{\text{Constr}}\end{array}} \quad KT = \mathit{MostSpec}(\Gamma, KST, T_1' \ldots T_n')$$

Fig. 35. Methods and constructors.

$$(90\text{-[Jam]})\quad \frac{\begin{array}{l}\Gamma, C \vdash K_1\diamond_{\text{Constr}}\\ \ldots\\ \Gamma, C \vdash K_n\diamond_{\text{Constr}}\\ \Gamma, C \vdash Me_1\diamond_{\text{Meth}}\\ \ldots\\ \Gamma, C \vdash Me_k\diamond_{\text{Meth}}\end{array}}{\Gamma \vdash \mathtt{class}\ C\ \{KS\ MS\}\diamond_{\text{Body-Decl}}} \quad \begin{array}{l}[C \notin \mathit{StandardClasses}(\Gamma)]\\ KS = \{K_1, \ldots, K_n\}\\ MS = \{Me_1, \ldots, Me_k\}\\ n, k \geq 0\end{array}$$

$$(91\text{-Jam})\quad \frac{\begin{array}{l}\Gamma \vdash C \lhd_m M\\ \Gamma \vdash \mathtt{mixin}\ M\ _\diamond_{\text{Body-Decl}}\\ \Gamma, C \vdash K_1\diamond_{\text{Constr}}\\ \ldots\\ \Gamma, C \vdash K_n\diamond_{\text{Constr}}\end{array}}{\Gamma \vdash \mathtt{class}\ C\ \{KS\ \emptyset\}\diamond_{\text{Body-Decl}}} \quad \begin{array}{l}KS = \{K_1, \ldots, K_n\}\\ n \geq 0\end{array}$$

$$(92\text{-Jam})\quad \frac{\begin{array}{l}\Gamma, M \vdash Me_1\diamond_{\text{Meth}}\\ \ldots\\ \Gamma, M \vdash Me_k\diamond_{\text{Meth}}\end{array}}{\Gamma \vdash \mathtt{mixin}\ M\ \{MS\}\diamond_{\text{Body-Decl}}} \quad \begin{array}{l}MS = \{Me_1, \ldots, Me_k\}\\ n, k \geq 0\end{array}$$

$$(93)\quad \frac{\Gamma \vdash BD_1\ \diamond_{\text{Body-Decl}} \ldots \Gamma \vdash BD_n\diamond_{\text{Body-Decl}}}{\Gamma \vdash \{BD_1 \ldots BD_n\}\diamond_{\text{Body-Decls}}} \quad n \geq 0$$

Fig. 36. Body declarations.

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

A simple way to get "for free" the property that a mixin instance should be a subtype of both the mixin and the parent type is to translate a mixin declaration into an interface declaration, and every instantiation into 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 Section 2.2.4), but only need to be copied at every instantiation).

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 one 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 `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. Accessor names depend on the name of the mixin where the corresponding fields were declared since otherwise in a case like that below

```
mixin M1 {
  int f ;
}
mixin M2 {
  boolean f ;
}
class C1 = M1 extends Object {}
class C2 = M2 extends C1 {}
```

hiding of fields would be translated into illegal overriding of accessor methods.

Note that the same is not needed for the field names in the translation since in this case hiding is correctly translated to hiding.

Another requirement to be met is that the translation must correctly simulate the Jam extended rule for overloading resolution (an inherited method in a mixin `M` must be considered as if it had been declared in a "generic" superclass of `M`, and hence considered less specific of a defined method with the same signature).

To this end, it is necessary to translate the declaration of a mixin `M` by *two* interfaces; the former, named `M`, corresponding to the mixin type, and the latter, named `Parent$M`, corresponding to the type *Parent*(`M`) introduced in Section 2.4, that is, the methods type of the generic parent on which the mixin can be instantiated.

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

Fig. 37. Translation of a mixin declaration.

Altogether, the two interface declarations `M` and `Parent$M` corresponding to the declaration of a mixin `M` will contain the following:

—the headings of all the instance methods declared in the mixin, either defined (in `M`) or inherited (in `Parent$M`);

—in `M`, the headings of a pair of getter and setter methods for every field declared in the mixin, either defined or inherited; in case of hidden (that is, both inherited and defined) fields only the accessor for the defined field will be present.

Moreover, the interface `M` will extend `Parent$M` and all the interfaces implemented by the mixin.

In Figure 37 we illustrate how the translation works in practice on the mixin `Undo` introduced in Section 2.1; an instantiation of the mixin `Undo` together with a simple test class is given in Figure 38, and the corresponding translation in Figure 39.

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.

As shown in Figure 39 in the translation of the method `main`, a field access on an expression of a mixin type is translated using an accessor; the only exceptions are field accesses through `this` which can be kept as they stand, as, for instance, in the translation of methods in class `ExampleWithUndo`. Note that accessors are not needed when the field access is on an expression of a type which is not a mixin type, including 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) ;
```

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

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

class ExampleWithUndo = Undo extends Example {}

public class Test {
   public static void main(String [] args) {
      Undo u = new ExampleWithUndo() ;
      u.setText("Hello ") ;
      u.setText("world !") ;
      u.undo() ;
      System.out.print(u.getText()) ;
      u.undo() ;
      System.out.println(u.getText()) ;
      System.out.println("u.lastText="+u.lastText) ;
   }
}
```

Fig. 38. `Undo` instantiation example.

accessors is generated in the `M` interface. Note that accessors for inherited fields are not declared in `Parent$M` since, analogously to the situation illustrated above, hiding an inherited by a defined field would be translated into an illegal overriding as shown, for instance, by the following mixin declaration:

```
mixin M {
   inherited boolean f ;
   int f ;
}
```

4.1.2 *Static Fields.* As shown in Section 2.2.4, static members 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

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

class ExampleWithUndo extends Example implements Undo {
  // Field "lastText"
  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) ;
  }
}

class Test {
  static void main(String[] args) {
    Undo u = new ExampleWithUndo() ;
    u.setText("Hello ") ;
    u.setText("world !") ;
    u.undo() ;
    System.out.print(u.getText()) ;
    u.undo() ;
    System.out.println(u.getText()) ;
    // Note the 'field' lastText access:
    System.out.println("u.lastText="+u.Undo_$get$_lastText()) ;
  }
}
```

Fig. 39. Translation of a mixin instantiation.

Jam. Second, we get the soundness of the Jam type system from the soundness of the Java type system [Drossopoulou and Eisenbach 1999] and Theorem 4.1, 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, and hence for type assertions.

4.2.1 *Translation of Environments and Related Judgments.* We denote by $[\![\Gamma]\!]$ the translation of a Jam environment Γ.

$$
\begin{array}{ll}
(1) & [\![\Gamma \vdash \gamma]\!] = [\![\Gamma]\!] \vdash [\![\gamma]\!]_\Gamma \\
(2) & [\![\Gamma]\!] = [\![\Gamma]\!]_\Gamma \\
(3) & [\![\gamma_1, \ldots, \gamma_n]\!]_\Gamma = \bigcup_{i \in \{1,\ldots,n\}} [\![\gamma_i]\!]_\Gamma
\end{array}
$$

Fig. 40. Translation of environments.

$$
\begin{array}{ll}
(4) & [\![M \lhd_i^1 I]\!]_\Gamma = \{M <_i^1 I\} \\
(5) & [\![M \lhd_i I]\!]_\Gamma = \{M \leq_i I\} \\
(6) & [\![C \lhd_m M]\!]_\Gamma = \{C \lhd_i^1 M\} \\
(7) & [\![M \text{ is}_m\ FST_d\ MST_d\ \texttt{inherited}\ FST_i\ MST_i]\!]_\Gamma = \\
& \{M \text{ is}_i\ \mathit{ToAbstract}(MST_d \cup \mathit{AccessorHeadings}(FST_i[FST_d], M)), \\
& \mathit{Parent}(M) \text{ is}_i\ \mathit{ToAbstract}(MST_i), \\
& M <_i^1 \mathit{Parent}(M)\} \\
(8) & [\![C \text{ is}_c\ K\ KST\ \emptyset\ \emptyset]\!]_\Gamma = \{C \text{ is}_c\ K\ KST\ FST_d\ MST_d \cup \mathit{AccessorHeadings}(FST_i[FST_d], M)\} \\
& \text{if } \Gamma \vdash C\ \lhd_m\ M, \Gamma \vdash M \text{ is}_m\ FST_d\ MST_d\ \texttt{inherited}\ FST_i\ MST_i \\
(9) & [\![FST_d\ MST_d\ \texttt{inherited}\ FST_i\ MST_i \diamond_{\text{MixinType}}]\!]_{\Gamma,M} = \\
& \{\mathit{ToAbstract}(MST_i) \diamond_{\text{InterfaceType}}, \\
& \mathit{ToAbstract}(MST_d \cup \mathit{AccessorHeadings}(FST_i[FST_d], M)) \diamond_{\text{InterfaceType}}\} \\
& \text{if } \Gamma \vdash M \text{ is}_m\ FST_d\ MST_d\ \texttt{inherited}\ FST_i\ MST_i \\
(10) & [\![C\ :\ FST_c\ MST_c]\!]_\Gamma = \{C : FST_c\ \ MST_c \cup \mathit{AllAccessorHeadings}_{[\![\Gamma]\!]}(C)\} \\
(11) & [\![M\ :\ FST_m\ MST_m]\!]_\Gamma = \{ M :\ \ \emptyset\ \mathit{ToAbstract}(MST_m \cup \\
& \qquad \mathit{AccessorHeadings}(FST_m, M)), \\
& \qquad \mathit{Parent}(M) : \emptyset\ \mathit{ToAbstract}(MST_i)\} \\
& \text{if } \Gamma \vdash M \text{ is}_m\ _\ _\ \texttt{inherited}\ _\ MST_i
\end{array}
$$

Fig. 41. Translation of type assertions.

Since assertions in Γ may be mutually recursive, analogously to what happens for the static semantics, the translation of Γ (Figure 40) uses an auxiliary function taking an additional argument which is a larger environment.

The translation of a Jam type assertion is a set of Java type assertions, and is defined in Figure 41. The functions *AllAccessorHeadings*, *AccessorHeadings*, *Accessors*, and *ToAbstract* are defined in Figure 42. Note that, differently from the function *AccessorHeadings*, the function *Accessors* returns both heading and body of accessors. For all the type assertions γ for which there is no translation clause, we implicitly assume that

$$[\![\gamma]\!]_\Gamma = \{\gamma\}.$$

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 these implementation assertions are translated into subinterface assertions (4–5); otherwise (that is, if T is a class) they remain the same. The instantiation assertion becomes an implementation assertion (6). A mixin declaration is transformed into the declaration of two interfaces, the former, where accessors are introduced, being a subinterface of the latter (7). A class declaration C is modified only in the case C is an instance of a mixin M; in this case the translation corresponds to the copy principle (8). A well-formed mixin type is translated into the two corresponding well-formed interface types (9); note that in this clause the translation function is indexed by two parameters: Γ, as usual, and M. This last parameter is

$$
\begin{aligned}
&\mathit{AllAccessorHeadings}_\Gamma(C) = \mathit{AllAccessorHeadings}_\Gamma(C') \\
&\quad \text{if } C \in \mathit{StandardClasses}(\Gamma), \Gamma \vdash C <_c^1 C' \\
&\mathit{AllAccessorHeadings}_\Gamma(C) = \mathit{AllAccessorHeadings}_\Gamma(C') \cup \mathit{AccessorHeadings}(FST_m, M) \\
&\quad \text{if } \Gamma \vdash C \lhd_m M, \Gamma \vdash C <_c^1 C', \Gamma \vdash M : FST_m \text{ _} \\
&\mathit{AllAccessorHeadings}_\Gamma(\texttt{Object}) = \emptyset \\
\\
&\mathit{AllAnnotatedAccessorHeadings}_\Gamma(C) = \mathit{AllAnnotatedAccessorHeadings}_\Gamma(C') \\
&\quad \text{if } C \in \mathit{StandardClasses}(\Gamma), \Gamma \vdash C <_c^1 C' \\
&\mathit{AllAnnotatedAccessorHeadings}_\Gamma(C) = \\
&\quad \mathit{AllAnnotatedAccessorHeadings}_\Gamma(C')[\mathit{Annotate}(M, \mathit{AccessorHeadings}(FST_m, M))]_{[\![\Gamma]\!]} \\
&\quad \text{if } \Gamma \vdash C \lhd_m M, \Gamma \vdash C <_c^1 C', \Gamma \vdash M : FST_m \text{ _} \\
&\mathit{AllAnnotatedAccessorHeadings}_\Gamma(\texttt{Object}) = \emptyset \\
\\
&\mathit{AccessorHeadings}(\{f_1 : FT_1, \ldots, f_n : FT_n\}, M) = \bigcup_{i \in \{1,\ldots,n\}} \mathit{AccessorHeading}(f_i : FT_i, M) \\
&\mathit{AccessorHeading}(f : \texttt{static } ST, M) = \emptyset \\
&\mathit{AccessorHeading}(f : \texttt{instance } ST, M) = \{M_\texttt{\$get\$_}f, \epsilon{:}\texttt{instance } ST \texttt{ throws } \emptyset, \\
&\qquad\qquad M_\texttt{\$set\$_}f, ST{:}\texttt{instance } ST \texttt{ throws } \emptyset\ \} \\
\\
&\mathit{Accessors}(\{f_1 : FT_1, \ldots, f_n : FT_n\}, M) = \mathit{Accessors}(f_1 : FT_1, M) \\
&\qquad\qquad \ldots \\
&\qquad\qquad \mathit{Accessors}(f_n : FT_n, M) \\
&\mathit{Accessors}(f : \texttt{static } T, M) = \epsilon \\
&\mathit{Accessors}(f : \texttt{instance } T, M) = \texttt{T } M_\texttt{\$get\$_}f\texttt{() \{ return } f\texttt{; \}} \\
&\qquad\qquad \texttt{T } M_\texttt{\$set\$_}f\texttt{(T n) \{ return } f\texttt{=n; \}} \\
\\
&\mathit{ToAbstract}(\{msig_1 : MT_1, \ldots, msig_n : MT_n\}) = \bigcup_{i \in \{1,\ldots,n\}} \mathit{ToAbstract}(msig_i : MT_i) \\
&\mathit{ToAbstract}(msig : \texttt{static } RT \texttt{ throws } ET) = \emptyset \\
&\mathit{ToAbstract}(msig : \texttt{instance } RT \texttt{ throws } ET) = \{msig : \texttt{abstract } RT \texttt{ throws } ET\} \\
&\mathit{ToAbstract}(msig : \texttt{abstract } RT \texttt{ throws } ET) = \{msig : \texttt{abstract } RT \texttt{ throws } ET\}
\end{aligned}
$$

Fig. 42. Definitions of auxiliary functions.

$$
(12)\ [\![\, BDS \,]\!]_\Gamma = [\![\, \mathit{ClassBodyDecls}(BDS) \,]\!]_{\Gamma, \mathit{MixinBodyDecls}(BDS)}
$$

Fig. 43. Translation of a set of body declarations.

the name that should be used to generate the names of accessors. Finally, type assignments for mixin instances and mixins are translated into type assignments for classes and interfaces, respectively, where accessors are introduced (10–11). Of course, we assume that there are no name conflicts between accessors and user defined methods.

4.2.2 *Translation of Body Declarations and Related Judgments.* The translation of a set of body declarations *BDS* consists of the translation of the class body declarations, *ClassBodyDecls*(*BDS*), which is defined with respect to an additional parameter which is the set of all the mixin body declarations, *MixinBodyDecls*(*BDS*), needed for translating mixin instances, as shown in Figure 43.

The translation of class body declarations is inductively defined by using auxiliary translation functions, one for each kind of subcomponent.

$$(13)\ [\![\,\texttt{class}\ C\ \{KS\ \emptyset\}\,]\!]_{\Gamma,MBD} = \texttt{class}\ C\ \{\,[\![\,KS\,]\!]_{\Gamma,C}\ [\![\,MBD(M)\,]\!]_{\Gamma,C} \cup \mathit{Accessors}(FST, M)\}$$
$$\text{if } \Gamma \vdash C \lhd_m M, \Gamma \vdash M : FST\ _$$

$$(14)\ [\![\,K\ RT\ \texttt{id}(ST_1\ \texttt{id}_1, \ldots, ST_n\ \texttt{id}_n)\ \texttt{throws}\ ET\ MB\,]\!]_{\Gamma,T} = K\ RT\ \texttt{id}(ST_1\ \texttt{id}_1, \ldots, ST_n\ \texttt{id}_n)\ \texttt{throws}\ ET\ [\![\,MB\,]\!]_{\Pi,\Gamma,T,K}$$
$$\text{where } \Pi = \{ST_1\ \texttt{id}_1, \ldots, ST_n\ \texttt{id}_n\}$$

$$(15)\ [\![\,\texttt{id}(ST_1\ \texttt{id}_1, \ldots, ST_m\ \texttt{id}_m)\ \texttt{throws}\ ET\ \{\texttt{super}(E_1, \ldots, E_n); SS\}\,]\!]_{\Gamma,T} =$$
$$\texttt{id}(ST_1\ \texttt{id}_1, \ldots, ST_m\ \texttt{id}_m)\ \texttt{throws}\ ET\ \{\ \texttt{super}(\,[\![\,E_1\,]\!]_{\Pi,\Gamma,T,\texttt{constructor}}, \ldots, [\![\,E_n\,]\!]_{\Pi,\Gamma,T,\texttt{constructor}}\,);\ [\![\,SS\,]\!]_{\Pi,\Gamma,T,\texttt{constructor}};\ \}$$
$$\text{where } \Pi = \{ST_1\ \texttt{id}_1, \ldots, ST_m\ \texttt{id}_m\}$$

$$(16)\ [\![\,E.f_i\,]\!]_{\Pi,\Gamma,T,K} = \begin{cases} [\![\,E\,]\!]_{\Pi,\Gamma,T,K}.M_\texttt{\$get\$}_f_i() & \text{if } \Pi,\Gamma,T,K \vdash E : \langle M, _\rangle \\ [\![\,E\,]\!]_{\Pi,\Gamma,T,K}.f_i & \text{otherwise} \end{cases}$$

$$(17)\ [\![\,E_1.f_i = E_2\,]\!]_{\Pi,\Gamma,T,K} = \begin{cases} [\![\,E_1\,]\!]_{\Pi,\Gamma,T,K}.M_\texttt{\$set\$}_f_i(\,[\![\,E_2\,]\!]_{\Pi,\Gamma,T,K}) & \text{if } \Pi,\Gamma,T,K \vdash E_1 : \langle M, _\rangle \\ [\![\,E_1\,]\!]_{\Pi,\Gamma,T,K}.f_i = [\![\,E_2\,]\!]_{\Pi,\Gamma,T,K} & \text{otherwise} \end{cases}$$

Fig. 44. Translation of body declarations.

The clauses defining the translation are explicitly given only in case the translation is not trivial (that is, the identity on terminals and the homomorphic extension on nonterminals). Such cases are shown in Figure 44.

In (13), the body of a mixin instance is translated into a class body which contains the translation of the constructors (provided at the moment of the instantiation) and of the methods contained in the mixin, plus the accessors for the fields. The translation of methods and constructors takes as additional parameter, besides Γ, the name C of the class where the method/constructor was declared.

In (14) and (15), we define the translation of methods and constructors. The translation of expressions and statements takes as additional parameters, besides Γ and C, a local environment Π and the kind K of the method/constructor where the expression/statement is placed.

In (16) and (17) a field access (respectively, an assignment to a field) is translated into the invocation of the corresponding getter (respectively, setter).

The translation of judgments of the form either $\Gamma, T \vdash \gamma$ or $\Pi, \Gamma, T, K \vdash \gamma$ (defined in Figure 20) is shown in Figure 45.

Note that judgments stating that some type assertion holds in a mixin M are translated into a set of judgments stating that (the translation of) the same type assertion holds in every instance of M. The additional parameter C in $[\![\gamma]\!]_{\Gamma,C}$ is needed because a type assertion of the form `this`: $\langle M, ET\rangle$, with M mixin, must be translated into `this`: $\langle C, ET\rangle$ into a given instance C of M (see clause 24 and the informal discussion in Section 2.5).

The last three clauses in Figure 45 show the nontrivial cases of the translation of judgments related to body declarations (those defined in Figure 20).

In (22–23) the annotated types of mixin instances and mixins are translated into annotated types for the corresponding class (respectively, interface) where accessors are introduced.

$$
\begin{array}{ll}
(18) & [\![\Gamma, C \vdash \gamma]\!] = [\![\Gamma]\!], C \vdash [\![\gamma]\!]_{\Gamma,C} \\
(19) & [\![\Gamma, M \vdash \gamma]\!] = \{ [\![\Gamma]\!], C \vdash [\![\gamma]\!]_{\Gamma,C} \mid \Gamma \vdash C \lhd_m M \} \\
(20) & [\![\Pi, \Gamma, C, K \vdash \gamma]\!] = \Pi, [\![\Gamma]\!], C, K \vdash [\![\gamma]\!]_{\Gamma,C} \\
(21) & [\![\Pi, \Gamma, M, K \vdash \gamma]\!] = \{ \Pi, [\![\Gamma]\!], C, K \vdash [\![\gamma]\!]_{\Gamma,C} \mid \Gamma \vdash C \lhd_m M \} \\
(22) & [\![C \ :^+ \ AMST]\!]_\Gamma = C :^+ AMST \cup \mathit{AllAnnotatedAccessorHeadings}_{[\![\Gamma]\!]}(C) \\
(23) & [\![M \ :^+ \ AMST]\!]_\Gamma = \{ M :^+ \ \mathit{ToAbstract}(AMST \cup \mathit{Annotate}(M, \\
& \qquad \mathit{AccessorHeadings}(FST_i[FST_d], M))) \\
& \qquad \mathit{Parent}(M) \ :^+ \ \mathit{Annotate}(\mathit{Parent}(M), \mathit{ToAbstract}(MST_i)) \} \\
& \text{if } \Gamma \vdash M \text{ is}_m \ FST_d \ MST_d \ \texttt{inherited} \ FST_i \ MST_i \\
(24) & [\![E : \langle T, ET \rangle]\!]_{\Gamma,C} = \begin{cases} [\![E]\!]_\Gamma : \langle T, ET \rangle & \text{if } E \neq \texttt{this} \\ [\![E]\!]_\Gamma : \langle C, ET \rangle & \text{if } E = \texttt{this} \end{cases}
\end{array}
$$

Fig. 45. Translation of judgments related to body declarations.

In (24) expressions keep their types under translation, except for the case of `this` expressions which change their type to `C` when translated with respect to an instance `C` of a mixin.

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. This is illustrated by the following diagram:

$$
\begin{array}{ccc}
Jam & \xrightarrow{[\![\,]\!]} & Java \\
\uparrow & \nearrow_{Id} & \\
Java & &
\end{array}
$$

More importantly, the translation preserves the static semantics, in the sense that a well-formed Jam program is translated into a well-formed Java program. More precisely, since we have represented a program as a pair consisting of an environment Γ and a set *BDS* of body declarations, this statement means that:

—a well-formed Jam environment Γ is translated into a well-formed Java environment $[\![\Gamma]\!]$;

—a set *BDS* of Jam body declarations (consisting of a set *MBDS* of mixin body declarations and a set *CBDS* of class body declarations) well-formed with respect to an environment Γ is translated to a set $[\![CBDS]\!]_{\Gamma,\,MBDS}$ of class body declarations well-formed with respect to $[\![\Gamma]\!]$ (recall that the set *MBDS* of mixin body declarations is not directly translated, but used as an additional parameter in the translation, needed for translating mixin instances).

THEOREM 4.1. *For each Γ Jam environment and BDS set of body declarations*

(i) *if $\Gamma \vdash \diamond$ is valid, then $[\![\Gamma]\!] \vdash \diamond$ is valid,*

(ii) *under the assumption that $\Gamma \vdash \diamond$ is valid, if $\Gamma \vdash BDS\diamond_{Body\text{-}Decls}$ is valid, then $[\![\Gamma \vdash BDS\diamond_{Body\text{-}Decls}]\!]$ is valid.*

The proof of Theorem 4.1 requires the following lemmas.

LEMMA 4.2

(i) *If* $\Gamma \vdash MST\diamond_{MethsType}$ *is valid, then* $\Gamma \vdash ToAbstract(MST)\diamond_{MethsType}$ *is valid.*
(ii) *If* $\Gamma \vdash MST\diamond_{MethsType}$, $\Gamma \vdash MST'\diamond_{MethsType}$ *are valid and* $Dom(MST) \cap Dom(MST') = \emptyset$, *then* $\Gamma \vdash MST \cup MST'\diamond_{MethsType}$ *is valid.*
(iii) *If* $\Gamma \vdash FST\diamond_{FieldsType}$ *is valid, then* $\Gamma \vdash AccessorHeadings(FST, M)\diamond_{MethsType}$ *is valid.*

PROOF. Easy check. □

LEMMA 4.3. *If* $\Gamma \vdash \diamond$, *then:*

(i) for $I \in Interfaces(\Gamma)$:
$ParentInterfaces(\Gamma, I) = ParentInterfaces([\![\Gamma]\!], I)$;
(ii) for $C \in StandardClasses(\Gamma)$:
$DImplementedInterfaces(\Gamma, C) = DImplementedInterfaces([\![\Gamma]\!], C)$;
for $C \notin StandardClasses(\Gamma)$:
$\{M\} \cup DImplementedInterfaces(\Gamma, C) = DImplementedInterfaces([\![\Gamma]\!], C)$;
(iii) for $M \in Mixins(\Gamma)$:
$\{Parent(M)\} \cup DImplementedInterfaces(\Gamma, M) = ParentInterfaces([\![\Gamma]\!], M)$.

PROOF

(i) We have to prove that $I <^1_i I' \in \Gamma$ *iff* $I <^1_i I' \in [\![\Gamma]\!]$. The "$\subseteq$" inclusion is trivial. For the "$\supseteq$" inclusion, the only translation clauses that produce $I <^1_i I' \in [\![\Gamma]\!]$ are either the identity translation from $I <^1_i I'$ or (4) from $I \lhd^1_i I'$, under the hypothesis that $I \in Mixins(\Gamma)$; but the latter case cannot happen since $I \in Interfaces(\Gamma)$.
(ii) We have to prove that $C \lhd^1_i I \in \Gamma$ *iff* $C \lhd^1_i I \in [\![\Gamma]\!]$. Analogously to the previous point, the "$\subseteq$" inclusion is trivial; for the "$\supseteq$" inclusion, the only translation clauses that produce $C \lhd^1_i I \in [\![\Gamma]\!]$ are either the identity translation from $C \lhd^1_i I$ or (6) from $C \lhd_m I$, but the latter case cannot happen since $C \in StandardClasses(\Gamma)$.
The proof for $C \notin StandardClasses(\Gamma)$ is analogous.
(iii) We have to prove that

$$(I = Parent(M) \text{ or } M \lhd^1_i I \in \Gamma) \textit{ iff } M <^1_i I \in [\![\Gamma]\!].$$

The "$\subseteq$" inclusion holds since $M <^1_i Parent(M)$ by translation clause (7) and $M \lhd^1_i I \in \Gamma \Rightarrow M <^1_i I \in [\![\Gamma]\!]$ by translation clause (4).
The "$\supseteq$" inclusion holds since the only translation clauses that can produce $M <^1_i I \in [\![\Gamma]\!]$ are either (7) with $I = Parent(M)$ or (4) from $M \lhd^1_i I$ or the identity translation from $M <^1_i I$, under the hypothesis that $M \in Interfaces(\Gamma)$, but the last case cannot happen since $M \in Mixins(\Gamma)$. □

LEMMA 4.4. *If* $\Gamma \vdash \diamond$ *is valid, then*

(i) *If* $\Gamma \vdash T \leq T'$ *is valid, then* $[\![\Gamma]\!] \vdash T \leq T'$ *is valid.*
(ii) $MST \overset{\Gamma}{\oplus} MST' = MST \overset{[\![\Gamma]\!]}{\oplus} MST'$ *and* $MST\ [MST']_\Gamma = MST\ [MST']_{[\![\Gamma]\!]}$,

(iii) $MostSpec(\Gamma, KST, AT) = MostSpec([\![\Gamma]\!], KST, AT)$,

(iv) $MostSpec(\Gamma, AMST, m, AT) = MostSpec([\![\Gamma]\!], AMST, m, AT)$,

(v) $\forall id : \Gamma(id) = \bot \Leftrightarrow [\![\Gamma]\!](id) = \bot$,

(vi) $ET \overset{\Gamma}{\ominus} ET' = ET \overset{[\![\Gamma]\!]}{\ominus} ET'$.

PROOF. Easy check. □

LEMMA 4.5. *If* $\Gamma \vdash T\colon_{_} \{msig : MT\} \cup _$ *is valid, then* $\Gamma \vdash T\colon^{+} \{T'\ msig : MT\} \cup _$ *is valid, for some* T'.

PROOF. Easy check. □

PROOF OF THEOREM 4.1 (i). In order to prove the thesis, we need the stronger property that every valid Jam judgment of the form $\Gamma \vdash \gamma$ is translated into a set of valid Java judgments, as formally expressed below:

(*) If $\Gamma \vdash \gamma$ is valid, then $[\![\gamma]\!]_\Gamma$ is well-defined and $[\![\Gamma]\!] \vdash [\![\gamma]\!]_\Gamma$ is valid.[12]

First, we show that the thesis is a corollary of (*). Indeed, by translation clause (2), $[\![\Gamma]\!] = [\![\Gamma]\!]_\Gamma$ and $\Gamma \vdash \diamond$ is an abbreviation for $\Gamma \vdash \Gamma\diamond$. Hence, from (*) where we choose as judgment $\Gamma \vdash \Gamma\diamond$, it follows that $[\![\Gamma]\!] \vdash [\![\Gamma]\!]_\Gamma$ is valid, and hence $[\![\Gamma]\!]$ is a well-formed Java environment.

We prove now (*) by induction on the metarules defining the validity of judgments. Assume that $\Gamma \vdash \gamma$ is a valid Jam judgment obtained instantiating some metarule with premises $\Gamma \vdash \gamma_1, \ldots, \Gamma \vdash \gamma_n$ (hence these judgments are valid too). We have to prove that $[\![\Gamma]\!] \vdash [\![\gamma]\!]_\Gamma$ is valid, under the inductive hypothesis that $[\![\Gamma]\!] \vdash [\![\gamma_i]\!]_\Gamma$ is valid, for $i = 1, \ldots, n$. For most metarules the translation of both γ and $\gamma_1, \ldots, \gamma_n$ is the identity, and the side condition either does not depend on Γ or trivially holds for $[\![\Gamma]\!]$ too. In all these cases, by inductive hypothesis, we get that $[\![\Gamma]\!] \vdash \gamma_1, \ldots, [\![\Gamma]\!] \vdash \gamma_n$ are valid; hence the thesis follows instantiating the same metarule taking as environment $[\![\Gamma]\!]$. We consider now the metarules for which either the translation of $\gamma, \gamma_1, \ldots, \gamma_n$ is not always the identity or we have to prove that the side condition holds for $[\![\Gamma]\!]$ when it holds for Γ.

(1). We have to prove that $[\![\Gamma]\!] \vdash [\![\gamma]\!]_\Gamma$ is valid. This follows by the same metarule, which can be instantiated since $[\![\gamma]\!]_\Gamma \in [\![\Gamma]\!]$ by translation clause (3).

(3). Assume that $\Gamma \vdash C \lhd_m M$ is valid (the other case is trivial). By translation clause (8), we have to prove that $[\![\Gamma]\!] \vdash C \leq_c C$ is valid, under the hypothesis that $[\![\Gamma]\!] \vdash C \text{ is}_c\ CT'$ is valid, for some class type CT'. The thesis follows by the same metarule.

(8). Assume that $T \in Mixins(\Gamma)$ (the other case is trivial). By translation clauses (4) and (5), we have to prove that $[\![\Gamma]\!] \vdash T \leq_i I$ is valid, under the inductive hypothesis that $[\![\Gamma]\!] \vdash T <^1_i I$ is valid. The thesis follows by metarule (5).

(10). Assume that $T \in Mixins(\Gamma)$ (the other case is trivial). By translation clause (5), we have to prove that $[\![\Gamma]\!] \vdash T \leq_i I$ is valid, under the inductive hypotheses that $[\![\Gamma]\!] \vdash T \leq_i I'$ and $[\![\Gamma]\!] \vdash I' \leq_i I$ are valid. The thesis follows by metarule (7).

[12] That is, $[\![\Gamma]\!] \vdash \gamma'$ is valid for each $\gamma' \in [\![\gamma]\!]_\Gamma$.

(11). By translation clauses (5) and (6), we have to prove that $[\![\Gamma]\!] \vdash C \lhd_i I$ is valid, under the inductive hypotheses that $[\![\Gamma]\!] \vdash C \lhd_i^1 M$ and $[\![\Gamma]\!] \vdash M \leq_i I$ are valid. The thesis follows by metarules (8) and (10).

(14). By translation clause (7), we have to prove that $[\![\Gamma]\!] \vdash C \leq M$ is valid under the inductive hypothesis that $[\![\Gamma]\!] \vdash C \lhd_i^1 M$ is valid. The thesis follows by the proof tree below:

$$(17)\quad \cfrac{(8)\ \cfrac{[\![\Gamma]\!] \vdash C \lhd_i^1 M}{[\![\Gamma]\!] \lhd_i M}}{[\![\Gamma]\!] \vdash C \leq M}.$$

(17). Assume that $T \in Mixins(\Gamma)$ (the other case is trivial). By translation clause (5), we have to prove that $[\![\Gamma]\!] \vdash T \leq I$ is valid under the inductive hypothesis that $[\![\Gamma]\!] \vdash T \leq_i I$ is valid. The thesis follows by metarule (13).

(18). By translation clause (7), we have to prove that $[\![\Gamma]\!] \vdash M \leq M$ is valid under the inductive hypothesis that $[\![\Gamma]\!] \vdash M \text{ is}_i\ IT$ is valid, for some interface type IT. The thesis follow by the proof tree below:

$$(13)\quad \cfrac{(6)\ \cfrac{[\![\Gamma]\!] \vdash M \text{ is}_i\ IT}{[\![\Gamma]\!] \vdash M \leq_i M}}{[\![\Gamma]\!] \vdash M \leq M}.$$

(26). The proof is analogous to that of metarule (7).

(28). By translation clause (7), we have to prove that $[\![\Gamma]\!] \vdash M \diamond_{\text{RefType}}$ is valid under the inductive hypothesis that $[\![\Gamma]\!] \vdash M \text{ is}_i\ IT$ is valid, for some interface type IT. The thesis follows by metarule (27).

(45). By translation clause (9), we have to prove that

(t1) $[\![\Gamma]\!] \vdash ToAbstract(MST')\diamond_{\text{InterfaceType}}$

(t2) $[\![\Gamma]\!] \vdash ToAbstract(MST \cup AccessorHeadings(FST'[FST], M))\diamond_{\text{InterfaceType}}$

are valid, under the inductive hypotheses that

$$[\![\Gamma]\!] \vdash FST\ MST\diamond_{\text{ModuleType}}$$
$$[\![\Gamma]\!] \vdash FST'\ MST'\diamond_{\text{ModuleType}}$$

are valid. These last two judgments can only be deduced by metarule (42); hence

(h1) $[\![\Gamma]\!] \vdash FST\diamond_{\text{FieldsType}}$,

(h2) $[\![\Gamma]\!] \vdash FST'\diamond_{\text{FieldsType}}$,

(h3) $[\![\Gamma]\!] \vdash MST\diamond_{\text{MethsType}}$,

(h4) $[\![\Gamma]\!] \vdash MST'\diamond_{\text{MethsType}}$

are valid too. The thesis (t1) follows by metarule (44):

$$(44)\quad \frac{[\![\Gamma]\!] \vdash ToAbstract(MST')\diamond_{\text{MethsType}}}{[\![\Gamma]\!] \vdash ToAbstract(MST')\diamond_{\text{InterfaceType}}},$$

where the premise is valid by (h4) and Lemma 4.2 (i). The thesis (t2) follows by metarule (44) too:

$$(44)\quad \frac{[\![\Gamma]\!] \vdash \mathit{ToAbstract}(\mathit{MST} \cup \mathit{AccessorHeadings}(\mathit{FST}'[\mathit{FST}], M))\diamond_{\text{MethsType}}}{[\![\Gamma]\!] \vdash \mathit{ToAbstract}(\mathit{MST} \cup \mathit{AccessorHeadings}(\mathit{FST}'[\mathit{FST}], M))\diamond_{\text{InterfaceType}}},$$

where the premise is valid by (h1), (h2), (h3), and Lemma 4.2 (iii), (ii) (assuming that there are no conflicts between accessors and usual method names), and (i).

(46). We have to prove that the same metarule can be instantiated, deducing the same result, taking as environment $[\![\Gamma]\!]$, that is, that $\mathit{ParentInterfaces}([\![\Gamma]\!], I) = \{I_1, \ldots, I_n\}$. This follows by Lemma 4.3 (i) and Lemma 4.4 (ii).

(47). Set

$$\mathit{FST} = \mathit{FST}_i[\mathit{FST}_d],$$

$$\mathit{MST} = (\mathit{MST}_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} \mathit{MST}_n)[\mathit{MST}_i[\mathit{MST}_d]_\Gamma]_\Gamma,$$

$$A = \mathit{AccessorHeadings}(\mathit{FST}, M),$$

$$\widetilde{\mathit{MST}} = \mathit{ToAbstract}((\mathit{MST}_1 \overset{[\![\Gamma]\!]}{\oplus} \ldots \overset{[\![\Gamma]\!]}{\oplus} \mathit{MST}_n)[\mathit{MST}_i[\mathit{MST}_d]_{[\![\Gamma]\!]}]_{[\![\Gamma]\!]} \cup A).$$

By translation clause (11), we have to prove that

$$(\text{t1})\ [\![\Gamma]\!] \vdash \mathit{Parent}(M) : \emptyset\ \mathit{ToAbstract}(\mathit{MST}_i)$$

$$(\text{t2})\ [\![\Gamma]\!] \vdash M : \emptyset\ \widetilde{\mathit{MST}}$$

are valid, under the following inductive hypotheses:

$$(\text{h1})\ \ [\![\Gamma]\!] \vdash [\![M \text{ is}_m\ \mathit{MKT}]\!]_\Gamma,$$

$$(\text{h2})\ \ [\![\Gamma]\!] \vdash [\![\mathit{MKT}\diamond_{\text{MixinType}}]\!]_\Gamma,$$

$$(\text{h3})\ \ [\![\Gamma]\!] \vdash I_1 : \emptyset\ \mathit{MST}_1 \ldots [\![\Gamma]\!] \vdash I_n : \emptyset\ \mathit{MST}_n.$$

The thesis (t1) follows by metarule (46):

$$(46)\quad \frac{\begin{array}{l}[\![\Gamma]\!] \vdash \mathit{Parent}(M) \text{ is}_i\ \mathit{ToAbstract}(\mathit{MST}_i)\\ [\![\Gamma]\!] \vdash \mathit{ToAbstract}(\mathit{MST}_i)\diamond_{\text{InterfaceType}}\end{array}}{[\![\Gamma]\!] \vdash \mathit{Parent}(M) : \emptyset\ \mathit{ToAbstract}(\mathit{MST}_i)}.$$

Indeed, it easy to show that $\mathit{ParentInterfaces}([\![\Gamma]\!], \mathit{Parent}(M)) = \emptyset$; moreover, the first premise is valid by virtue of (h1) and translation clause (7), whereas the second is valid by (h2) and translation clause (9).

For the second thesis, set $\widehat{\mathit{MST}} = (\mathit{MST}_1 \overset{[\![\Gamma]\!]}{\oplus} \ldots \overset{[\![\Gamma]\!]}{\oplus} \mathit{MST}_n \overset{[\![\Gamma]\!]}{\oplus} \mathit{ToAbstract}(\mathit{MST}_i))[\mathit{ToAbstract}(\mathit{MST}_d \cup A)]_{[\![\Gamma]\!]}$. Using metarule (46), we get

$$(46)\quad \frac{\begin{array}{l}[\![\Gamma]\!] \vdash M \text{ is}_i\ \mathit{ToAbstract}(\mathit{MST}_d \cup A)\\ [\![\Gamma]\!] \vdash \mathit{ToAbstract}(\mathit{MST}_d \cup A)\diamond_{\text{InterfaceType}}\\ [\![\Gamma]\!] \vdash \mathit{Parent}(M) : \emptyset\ \mathit{ToAbstract}(\mathit{MST}_i)\\ [\![\Gamma]\!] \vdash I_1 : \emptyset\ \mathit{MST}_1 \ldots [\![\Gamma]\!] \vdash I_n : \emptyset\ \mathit{MST}_n\end{array}}{[\![\Gamma]\!] \vdash M : \emptyset\ \widehat{\mathit{MST}}}.$$

Indeed, $ParentInterfaces([\![\Gamma]\!], M) = \{I_1, \ldots, I_n, Parent(M)\}$ by Lemma 4.3 (iii). Moreover, the premises are valid, respectively, by

(h1) and translation clause (7),
(h2) and translation clause (9),
(t1) which we have already proved, and
(h3).

It is easy to see that $\widetilde{MST}$ is defined by the validity of $\Gamma \vdash M : FST\ MST$ and by Lemma 4.2 (ii), (i). Now, we have to show that $\widehat{MST}$ is defined whenever $\widetilde{MST}$ is defined (hence metarule (46) can be instantiated as shown above) and that, in such cases, they are equal. The first part can be proved showing, by case analysis, that if $\widehat{MST}$ is not defined, then $\widetilde{MST}$ is undefined too. Analogously, the second part can be proved showing, by case analysis, that each method in $\widehat{MST}$ is also in $\widetilde{MST}$ with the same type, and conversely. Note that for proving these two parts we can use the fact that $\overset{\Gamma}{\otimes}$ is the greatest lower bound with respect to the relation $\leq_e$ (see Ancona et al. [2000] for a formal proof of this property).

(49). We have to prove that the same metarule can be instantiated, deducing the same result, taking as environment $[\![\Gamma]\!]$, that is, that

$$DImplementedInterfaces([\![\Gamma]\!], C) = \{I_1, \ldots, I_n\}.$$

This follows by Lemma 4.3 (ii) and Lemma 4.4 (ii).

(50). Set

$$\begin{aligned} A &= AccessorHeading(FST_i[FST_d], M), \\ FST_c &= FST'[FST_d], \\ MST_c &= (MST_1 \overset{[\![\Gamma]\!]}{\oplus} \ldots MST_n \overset{[\![\Gamma]\!]}{\oplus} Abstract(MST')) \\ &\quad [NonAbstract(MST')[MST_d]_{[\![\Gamma]\!]}]_{[\![\Gamma]\!]}. \end{aligned}$$

By translation clauses (11) we have to prove that

$$\text{(t1)}\ [\![\Gamma]\!] \vdash C : FST_c\ MST_c \cup AllAccessorHeadings_{[\![\Gamma]\!]}(C)$$

is valid. To this aim consider the following inductive hypotheses, following from the corresponding hypotheses of metarule (50) and from translation clauses 8, 10, and 11, respectively:

(h1) $[\![\Gamma]\!] \vdash C \text{ is}_c\ K\ KST\ FST_d\ MST_d \cup A$,
(h2) $[\![\Gamma]\!] \vdash C' : FST'\ MST' \cup AllAccessorHeadings_{[\![\Gamma]\!]}(C')$,
(h3) $[\![\Gamma]\!] \vdash M : \emptyset\ ToAbstract(MST_m \cup A)$.

Note that the judgment $\Gamma \vdash M : FTS_m\ MST_m$ can only be deduced by metarule (47); therefore $FST_m = FST_i[FST_d]$ by Lemma 4.4 (ii).

Furthermore, the following inductive hypotheses are obtained by identity translation from the corresponding hypotheses of metarule (50):

(h4) $[\![\Gamma]\!] \vdash K\ KST\ \emptyset\ \emptyset \diamond_{\text{ClassType}}$,
(h5) $[\![\Gamma]\!] \vdash C <^1_c C'$,
(h6) $[\![\Gamma]\!] \vdash I_1 : \emptyset\ MST_1, \ldots, [\![\Gamma]\!] \vdash I_n : \emptyset\ MST_n$.

By Lemma 4.3 (ii)

$$\{M\} \cup DImplementedInterfaces(\Gamma, C) = DImplementedInterfaces([\![\Gamma]\!], C);$$

then applying metarule (49) to hypotheses (h1)–(h6) we obtain

$$\text{(t2)}\ [\![\Gamma]\!] \vdash C : FST'_c\ MST'_c,$$

where

$$\begin{aligned} FST'_c &= FST'[FST_d], \\ MST'_c &= (MST_1 \overset{[\![\Gamma]\!]}{\oplus} \ldots MST_n \overset{[\![\Gamma]\!]}{\oplus} ToAbstract(MST_m \cup A) \overset{[\![\Gamma]\!]}{\oplus} \\ &\quad [Abstract(MST' \cup A'))\ NonAbstract(MST' \cup A')[MST_d \cup A]_{[\![\Gamma]\!]}]_{[\![\Gamma]\!]}, \\ A' &= AllAccessorHeadings_{[\![\Gamma]\!]}(C'). \end{aligned}$$

We show now that $MST'_c = MST_c \cup AllAccessorHeadings_{[\![\Gamma]\!]}(C)$ so that (t1) and (t2) coincide. By hypothesis $\Gamma \vdash FST'\ MST' \leq_{mod} FST_i\ MST_i$ of metarule (50) and by metarules (24) and (23) and by definition of $\overset{[\![\Gamma]\!]}{\oplus}$, *ToAbstract* and *Abstract* we have $ToAbstract(MST_m \cup A) \overset{[\![\Gamma]\!]}{\oplus} Abstract(MST' \cup A') = ToAbstract(MST_m) \cup ToAbstract(A) \overset{[\![\Gamma]\!]}{\oplus} Abstract(MST') = ToAbstract(A) \overset{[\![\Gamma]\!]}{\oplus} Abstract(MST')$. Furthermore, by definition of update we have

$$\begin{aligned} MST'_c &= (MST_1 \overset{[\![\Gamma]\!]}{\oplus} \ldots MST_n \overset{[\![\Gamma]\!]}{\oplus} ToAbstract(A) \overset{[\![\Gamma]\!]}{\oplus} Abstract(MST')) \\ &\quad [NonAbstract(MST')[MST_d \cup A \cup A']_{[\![\Gamma]\!]}]_{[\![\Gamma]\!]} \\ &= (MST_1 \overset{[\![\Gamma]\!]}{\oplus} \ldots MST_n \overset{[\![\Gamma]\!]}{\oplus} Abstract(MST')) \\ &\quad [NonAbstract(MST')[MST_d]_{[\![\Gamma]\!]}]_{[\![\Gamma]\!]} \cup A \cup A'. \end{aligned}$$

Therefore we conclude by the fact that $AllAccessorHeadings_{[\![\Gamma]\!]}(C) = A \cup A'$.

Finally note that (t2) can be inferred since the last side condition of metarule (49) derives from the trivial to prove equalities $Kind(MST_c) = Kind(MST_c \cup AllAccessorHeadings_{[\![\Gamma]\!]}(C)) = Kind(MST'_c)$.

(52). By induction hypothesis, translation clauses (2), (3), and (10), and Lemma 4.4 (v), we can instantiate the same metarule.

(53). By induction hypothesis, translation clauses (2), (3), (10), (8), and (6), and Lemma 4.4 (v), we can instantiate metarule (52).

(54). By induction hypothesis, translation clauses (2) and (3), and Lemma 4.4 (v), we can instantiate the same metarule.

(55). By induction hypothesis and translation clauses (2), (4), and (11), the following judgments are valid:

(h1) $[\![\Gamma]\!] \vdash [\![\Gamma']\!]_\Gamma \diamond$,
(h2) $[\![\Gamma]\!] \vdash M : \emptyset\ ToAbstract(MST_m \cup AccessorHeadings(FST_m, M))$,
(h3) $[\![\Gamma]\!] \vdash Parent(M) : \emptyset\ ToAbstract(MST_i)$.

By Lemma 4.4 (v), $[\![\Gamma']\!]_\Gamma(\mathit{Parent}(M)) = \bot$; hence we can apply metarule (54) with hypotheses (h1) and (h3) to derive the validity of

$$(h4)[\![\Gamma]\!] \vdash [\![\Gamma']\!]_\Gamma \cup \{\mathit{Parent}(M) \text{ is}_i \ \mathit{ToAbstract}(\mathit{MST}_i)\} \diamond .$$

Clearly, metarule (54) can be applied again with hypotheses (h4) and (h2) to derive the validity of

$$\begin{aligned}[\![\Gamma]\!] \vdash [\![\Gamma']\!]_\Gamma \ \cup \ \{&\mathit{Parent}(M) \text{ is}_i \ \mathit{ToAbstract}(\mathit{MST}_i),\\ &M \ \text{is}_i \ \mathit{ToAbstract}(\mathit{MST}_d \cup \mathit{AccessorHeadings}(\mathit{FST}_i[\mathit{FST}_d], M)),\\ &M <_i^1 \mathit{Parent}(M), M <_i^1 I_1, \ldots, M <_i^1 I_n\}\diamond,\end{aligned}$$

which is the translation of the thesis of metarule (55), by virtue of translation clauses (4) and (7).

(56). We have to prove that the following judgments are valid:

$$\text{(t1) } [\![\Gamma]\!] \vdash M \ \leq \mathit{Parent}(M),$$
$$\text{(t2) } [\![\Gamma]\!] \vdash \mathit{Parent}(M) \leq \mathit{Parent}(M).$$

To this aim, by virtue of translation clause (7), the following induction hypotheses can be used:

$$\text{(h2) } [\![\Gamma]\!] \vdash M <_i^1 \mathit{Parent}(M),$$
$$\text{(h1) } [\![\Gamma]\!] \vdash \mathit{Parent}(M) \text{ is}_i \ \mathit{IT}'.$$

The judgment (t1) can be proved from (h1) by applying metarules (5) and (10), whereas (t2) can be proved from (h2) by applying metarules (6) and (13).

(58). Set

$$\begin{aligned}\mathit{AMST} = \ &(\mathit{AMST}_1 \overset{[\![\Gamma]\!]}{\oplus} \ldots \overset{[\![\Gamma]\!]}{\oplus} \mathit{AMST}_n)\\ &[\mathit{Annotate}(\mathit{Parent}(M), \mathit{MST}_i)[\mathit{Annotate}(M, \mathit{MST}_d)]_{[\![\Gamma]\!]}]_{[\![\Gamma]\!]}.\end{aligned}$$

By translation clauses (23), we have to prove that

$$\text{(t1) } [\![\Gamma]\!] \ \vdash \ \mathit{Parent}(M) :^+ \mathit{Annotate}(\mathit{Parent}(M), \mathit{ToAbstract}(\mathit{MST}_i)) \text{ and}$$
$$\begin{aligned}\text{(t2) } [\![\Gamma]\!] \ \vdash \ &M :^+ \mathit{ToAbstract}(\mathit{AMST} \cup \mathit{Annotate}(M,\\ &\mathit{AccessorHeadings}(\mathit{FST}_i[\mathit{FST}_d], M)))\end{aligned}$$

are valid, under the inductive hypotheses

$$\text{(h1) } [\![\Gamma]\!] \vdash M \ \text{is}_i \ \mathit{ToAbstract}(\mathit{MST}_d \cup \mathit{AccessorHeadings}(\mathit{FST}_i[\mathit{FST}_d], M)),$$
$$\text{(h2) } [\![\Gamma]\!] \vdash \mathit{Parent}(M) \text{ is}_i \ \mathit{ToAbstract}(\mathit{MST}_i),$$
$$\text{(h3) } [\![\Gamma]\!] \vdash I_1 \ :^+ \mathit{AMST}_1, \ldots [\![\Gamma]\!] \vdash I_n, :^+ \mathit{AMST}_n,$$

where (h1) and (h2) are obtained by translation clause (7). By applying metarule (57) with hypothesis (h2) (it is easy to see that $\mathit{ParentInterfaces}(\Gamma, \mathit{Parent}(M)) = \emptyset$), we derive the validity of (t1).

Now by Lemma 4.3 (iii) we have that

$$\mathit{ParentInterfaces}([\![\Gamma]\!], M) = \{\mathit{Parent}(M), I_1, \ldots, I_n\};$$

therefore we can apply metarule (57) again with hypotheses (h2), (h3), and (t1) and conclude that $[\![\Gamma]\!] \vdash M :^{+} \widehat{AMST}$ is valid, where

$$\widehat{AMST} =$$
$$(AMST_1 \overset{[\![\Gamma]\!]}{\oplus} \ldots \overset{[\![\Gamma]\!]}{\oplus} AMST_n \overset{[\![\Gamma]\!]}{\oplus} \mathit{Annotate}(\mathit{Parent}(M), \mathit{ToAbstract}(MST_i)))$$
$$[\mathit{Annotate}(M, \mathit{ToAbstract}(MST_d \cup \mathit{AccessorHeadings}(FST_i[FST_d], M)))]_{[\![\Gamma]\!]}.$$

Finally, we can prove the equality

$$\mathit{ToAbstract}(AMST \cup \mathit{Annotate}(M, \mathit{AccessorHeadings}(FST_i[FST_d], M))) = \widehat{AMST}$$

whenever the left-hand side is defined, analogously to what we have done in the proof of metarule (47) using Lemma 4.4 (ii).

(60). The proof is analogous to that of metarule (49).

(61). The proof is analogous to that of metarule (50). □

PROOF OF THEOREM 4.1 (ii) (SKETCH). Let us denote by *MBDS* and *CBDS*, respectively, the sets of mixin body declarations and class body declarations in *BDS*.

We have to prove that, for each *BD* class body declaration in *CBDS*, if $\Gamma \vdash BD\diamond_{\text{Body-Decl}}$ is valid, then $[\![\Gamma]\!] \vdash [\![BD]\!]_{\Gamma,MBDS}\diamond_{\text{Body-Decl}}$ is valid, under the assumption that $\Gamma \vdash MBDS\diamond_{\text{Body-Decls}}$ is valid.

This can be proved by induction on the metarules defining the validity of judgments, analogously to what we have done for Theorem 4.1 (i).

We consider below the metarules for which either the translation of premises/consequence is not always the identity or it is not trivial that the side condition holds for $[\![\Gamma]\!]$ whenever it holds for Γ.

(67). If $T \in \mathit{Classes}(\Gamma)$, then we have to prove that the same metarule can be instantiated taking as environment $[\![\Gamma]\!]$, that is, $KT = \mathit{MostSpec}([\![\Gamma]\!], \{KT_1, \ldots, KT_n\}, T_1 \ldots T_m)$. This follows by inductive hypotheses and Lemma 4.4 (iii). If $T \in \mathit{Mixins}(\Gamma)$, then we have to prove that the same metarule can be instantiated for each class instance of T. This follows from inductive hypothesis, the optional side condition and Lemma 4.4 (iii).

(68). We consider the most difficult case in which $T, T' \in \mathit{Mixins}(\Gamma)$ (the other cases are simpler). Set $AH = \mathit{AccessorHeadings}(\{f_1 : FT_1, \ldots, f_n : FT_n\}, T')$. There are two cases.

If $E \neq$ `this`, by translation clause (16), (21), and (20) we have to prove that, for all C such that $\Gamma \vdash C \lhd_m T$,

$$\Pi, [\![\Gamma]\!], C, K \vdash [\![E]\!]_{\Pi,\Gamma,C,K}.T'_\$\mathtt{get}\$_f_i(): \langle \mathit{Type}(FT_i), ET \rangle$$

is valid.

By induction hypotheses and translation clauses (11), (21), and (24) we have that the following judgments are valid:

$$\Pi, [\![\Gamma]\!], C, K \vdash [\![E]\!]_{\Pi,\Gamma,C,K}: \langle T', ET \rangle,$$
$$[\![\Gamma]\!] \vdash T' : \emptyset\ \mathit{ToAbstract}(AH \cup _).$$

By Lemma 4.5, $[\![\Gamma]\!] \vdash T'{:}^{+}\{_\ T'_\mathtt{\$get\$}_f_i : _\} \cup _$ is valid too. Hence, we can instantiate the metarule (73) to get the thesis.

If $E = \mathtt{this}$, by translation clause (16), (21), and (24) we have to prove that, for all C such that $\Gamma \vdash C \lhd_m T$,

$$\Pi, [\![\Gamma]\!], C, K \vdash [\![E]\!]_{\Pi,\Gamma,C,K} . f_i \colon \langle \mathit{Type}(FT_i), ET \rangle$$

is valid.

In this case it is easy to see that it must be $T' = T$ and, therefore, by induction hypotheses and translation clauses (11), (21), and (24) we have that the following judgments are valid:

$$\Pi, [\![\Gamma]\!], C, K \vdash [\![E]\!]_{\Pi,\Gamma,C,K} \colon \langle C, ET \rangle,$$
$$[\![\Gamma]\!] \vdash T : \emptyset\ \mathit{ToAbstract}(AH \cup _).$$

Now under the assumptions of the theorem, we have that from the validity of $[\![\Gamma]\!] \vdash T : \emptyset\ \mathit{ToAbstract}(AH \cup _)$ and $\Gamma \vdash C \lhd_m T$ we can derive the validity of $[\![\Gamma]\!] \vdash C : \{f_1 : FT_1, \ldots, f_n : FT_n\} \cup _\ _$ (easy check); hence we can apply metarule (68) to get the thesis.

(73), (75), (76). Using Lemma 4.4 (iv), we can instantiate the same metarule.

(78). The proof is analogous to that of metarule (68).

(83). Using Lemma 4.4 (vi), we can instantiate the same metarule.

(88). Assume that $T \in \mathit{Mixins}(\Gamma)$ (the other case is trivial). We have to prove that, for all C such that $\Gamma \vdash C \lhd_m T$,

$$[\![\Gamma]\!], C \vdash K\ RT\ \mathtt{id}\,(ST_1\ \mathtt{id}_1, \ldots, ST_n\ \mathtt{id}_n)\ \mathtt{throws}\ ET\ [\![MB]\!]_{\Pi,\Gamma,C,K} \diamond_{\mathrm{Meth}}$$

is valid, under the inductive hypotheses that

$$[\![\Gamma]\!] \vdash RT' \leq RT,$$
$$\Pi, \Gamma, C, K \vdash MB \colon \langle RT'', ET' \rangle$$

are valid, where $\Pi = \{ST_1\ \mathtt{id}_1, \ldots, ST_n\ \mathtt{id}_n\}$ and $RT'' = RT'$ if the return expression is not `this`, $RT'' = C$ otherwise.

We can instantiate the same metarule considering that only these two cases are possible. The case $RT'' = RT'$ is trivial; in the case $RT'' = C$ it is easy to see that it must be $RT' = T$; hence it suffices to note that $[\![\Gamma]\!] \vdash RT'' = C \leq RT' = T$ since $\Gamma \vdash C \leq T$ by metarule (14) and Lemma 4.4 (i), and $[\![\Gamma]\!] \vdash RT' \leq RT$ by inductive hypothesis. Hence, by metarule (19), $[\![\Gamma]\!] \vdash RT'' \leq RT$. □

Note that the converse of the previous theorem does not hold, that is, some illegal Jam program can be translated into a correct Java program. Formally, there exist Γ such that $\Gamma \vdash \diamond$ is not valid and $[\![\Gamma]\!] \vdash \diamond$ is valid. For instance, consider an environment where $E_1, E_2 \leq_c$ `Throwable` and $\Gamma \not\vdash E_1 \leq E_2$ and the following code fragment:

```
mixin M implements I {
  inherited void m() throws E1 ;
}
interface I{
  void m() throws E2();
}
```

Indeed, this declaration is illegal in Jam since the inherited method `m` overrides the method `m` in `I` but has an incompatible `throws` clause. However, when translating the mixin declaration, we get an interface `M` extending both interfaces `Parent$M` and `I`, and there is no check of `throws` clauses.

The fact that illegal Jam programs can be translated into correct Java programs implies, on the implementation side, that a Jam compiler cannot completely delegate the type checking to the underlying Java compiler.

5. IMPLEMENTATION

This section is devoted to the presentation of the Jam to Java translator. In developing the translator, our main aim was to provide a prototype based on a simple implementation strategy for Jam, driven by a formal translation. Hence the translator has some limits, which are discussed below.

The section is structured as follows. First, we briefly illustrate how to use the translator and discuss its drawbacks, also sketching some possible alternatives.

Then we go into more details and provide a running example for explaining how the translator handles two features that, for sake of simplicity, have been omitted in the formal definition of Jam given in the previous sections: packages and access modifiers. Also, some implementation details of the translation are discussed.

Finally, we briefly describes *Invisible Jacc*, the parser generator used for implementing the translator.

5.1 Jam Compiler (`jamc`)

The implementation of the Jam to Java translation is called `jamc` (Jam compiler); it is a command line utility which is used in the same way as an ordinary compiler, that is, it takes as arguments a sequence of Jam files (extension `.jam`) and produces a set of Java files. Since `jamc` has been implemented in Java, the corresponding command invokes a script, rather than a true executable file, which makes the Java Virtual Machine (JVM) load and execute the class file which implements `jamc`. All the arguments of the `jamc` script are passed to the JVM. Assume, for instance, that the following files have to be compiled:

—`Colored.jam` containing the declaration of the mixin `Colored`;
—`Point.jam` containing the declaration of the class `Point`;
—`ColoredPoint.jam` containing the declaration of the class `ColoredPoint`, obtained as instance of `Point` applied to `Colored`.

You can compile them typing[13]

```
jamc Colored.jam Point.jam ColoredPoint.jam
```

The order of arguments is immaterial.

The translator performs a complete syntactic analysis and only a partial type-checking in order to identify the expressions that need to be modified in the translation process. This means that every either lexical or syntactic error

[13]This example assumes the use of a command line shell.

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`. Therefore, `jamc` does not completely implement the static semantics defined in the previous sections. There is, however, a minimal core of checks that must be performed by the Jam compiler because some *incorrect* Jam program can be translated to a *correct* Java program, as already pointed out at the end of the preceding section.

In the current implementation, we have chosen to follow the formal translation, based on the copy principle, in a straightforward way, because this guarantees good performances. This means that a mixin instantiation is translated into a Java class obtained, roughly, by extending the parent by a copy of the definitions contained in the corresponding mixin declaration. In other words, following the terminology introduced by Pizza [Odersky and Wadler 1997], we have developed a *heterogeneous* translation which tends to favor the running time of the generated code penalizing the bytecode size and modularity. More precisely, there are the following drawbacks.

—The duplication of code which is avoided from the programmer's point of view is in a sense "delayed" to the implementation.
—More importantly, this approach is not compatible with separate compilation in the Java sense (a class can be compiled in a context where used classes are available only in bytecode format), since for translating a mixin instantiation we need the source code of the mixin.

Alternatively, a *homogeneous* translation could be considered in order to share as much as possible code between the instantiations of a mixin and to allow separate compilation of mixin instances, at the cost of some running time overhead. This translation would be preferred in environments like embedded systems and smart-card applications where the code size is often a main concern. We sketch this alternative translation in the next subsection.

Another solution could be an implementation not based on a translation in Java, but rather directly generating bytecode (as, for instance, is done in Jiazzi, a recent system for constructing and linking components in Java [McDirmid et al. 2001]). In this case separate compilation would be preserved; however, duplication of code would be in this case delayed to the bytecode level. Indeed, bytecode for a mixin instantiation would be obtained, roughly, by combining together the components corresponding to the mixin and to the parent, and hence by duplicating mixin's bytecode, analogously to what happens in Jiazzi when linking two components.

Another inconvenience of the current implementation is that some field accesses are translated into method invocations (see in the sequel), and this can impose performance penalties.

5.2 A Homogeneous Translation

A homogeneous translation for Jam (that is, a translation which does not duplicate mixin code for each instantiation) could be obtained by generating, as translation of a mixin `M`, in addition to an interface `M` as it currently is, a class,

say `BodyM`, which contains, for each method defined in the mixin, a static method with an additional parameter representing `this`.

More precisely, for each method `T m(T1 a1,...,Tn an){`*body*`}` declared in a mixin `M`, the corresponding class `BodyM` should contain a static method

`static T m(M This, T1 a1,...,Tn an){`$\underline{body}$`}`,

where $\underline{body}$ is obtained from *body* by:

—replacing every explicit or implicit reference to `this` by a reference to `This` (the added first parameter);
—replacing every field access *e*`.f` (with *e* of type `M`) with an invocation to the corresponding accessor;
—replacing every `super` field access/method invocation by an invocation to a corresponding accessor; indeed in the interface `M`, in addition to accessors for fields, we should add accessors for each inherited field or method.

In this way, a mixin instance `C` can be translated into an ordinary class which extends the declared superclass and implements the interface `M`, declaring all the fields and methods defined in the mixin. However, the method bodies are no longer a copy of method bodies in `M`, but only contain invocations (with first argument `this`) to the static methods which are contained in the class `BodyM`. Note that in this way translation of a mixin instance can be performed separately, that is, by only having the type information part of the corresponding mixin.

A real implementation of this translation schema should also consider other features. Static methods could be translated analogously by adding a `Class` parameter and using Java reflection. Access modifiers should be weakened in translating mixin instances in order to allow the class `BodyM` to access private members of `M`. Field initializers could be translated into invocations of static methods in `BodyM` as we do with instance method invocations.

5.3 Packages and Access Modifiers

Let us have a closer look at the three example files introduced in the previous section (Figures 46–48). In Figure 46, the declaration of the mixin `Colored` contains two features not considered in our theoretical treatment of Jam: packages and access modifiers. Indeed, `Colored` imports all classes/interfaces of the package `java.awt` and declares a `private` field.

Whereas the use of packages in mixins seems to be harmless, access modifiers need some care. For `private` fields, like `color` in the example, it is clear that they do not belong to the mixin type but are visible only within the mixin itself and, moreover, are copied as `private` fields in every mixin instance. However, all other nonpublic (that is, `protected` and `package`) components cannot belong to mixin types, at least with the current Java translation where these types are implemented as Java interfaces. Figure 48 contains the declaration of the class `ColoredPoint` obtained by instantiating the mixin `Colored` (Figure 46) on the class `Point`(Figure 47).

Two different constructors are provided for the class `ColoredPoint`: the default constructor plus a constructor which initializes the field `color` by means

```
import java.awt.* ;

mixin Colored {
   private Color color = Color.black ;
   public Color getColor() {
      return color ;
   }
   public void setColor(Color newColor) {
      color = newColor ;
   }
}
```

Fig. 46. Declaration of mixin `Colored`.

```
class Point {
   int x, y ;
}
```

Fig. 47. Declaration of class `Point`.

```
import java.awt.* ;

class ColoredPoint = Colored extends Point {
   ColoredPoint() {}
   ColoredPoint(Color c) {
      setColor(c) ;
   }
}
```

Fig. 48. Declaration of class `ColoredPoint`.

of the invocation `setColor(c)`. Figures 49–51 contain the Java files produced by `jamc` as translation of the respective Jam files.

Note that in Figure 49 there are no accessors for the field `color` of the mixin `Colored`. Finally, it is worth noting that the Java files produced by `jamc` contain no `import` directives; indeed, `import` is useless since `jamc` uses only fully qualified names.

5.4 Line Numbering

As already explained, Jam delegates most of the static checks to the Java compiler; therefore error messages quite often refer to the Java files produced by `jamc` rather than to the corresponding Jam files.

For this reason, `jamc` labels all statements and field declarations with references to the corresponding line in the original Jam file, so that errors recovering

```
//
// File automagically created from
// "Colored.jam" — DO NOT EDIT !!!
//
// (Jam may overwrite this file next time you'll run it)
//

interface Parent$Colored {

   // Methods:

}

public interface Colored extends Parent$Colored {

   // Methods:
   public java.awt.Color getColor() ;
   public void setColor(java.awt.Color newColor) ;

}
```

Fig. 49. File `Colored.java`.

```
//
// File automagically created from
// "Point.jam" — DO NOT EDIT !!!
//
// (Jam may overwrite this file next time you'll run it)
//

class Point {

   int x, y ; // file: Point.jam, line: 3

}
```

Fig. 50. File `Point.java`.

becomes easier. This is achieved by adding single line comments in the Java files containing a file name and a line number (see Figures 48 and 51). Note that this information is particularly useful when a mixin instance is obtained from a mixin and a parent class contained in different Jam files (as in Figure 51).

```
//
// File automagically created from
// "ColoredPoint.jam" — DO NOT EDIT !!!
//
// (Jam may overwrite this file next time you'll run it)
//

class ColoredPoint extends Point implements Colored {
   // Field "color"
   private java.awt.Color color = java.awt.Color.black ;

   public java.awt.Color getColor() {
      return color ; // file: Colored.jam, line: 6
   }

   public void setColor(java.awt.Color newColor) {
      color=newColor ; // file: Colored.jam, line: 9
   }

   ColoredPoint() {
   }

   ColoredPoint(java.awt.Color c) {
      setColor(c) ; // file: ColoredPoint.jam, line: 6
   }

}
```

Fig. 51. File `ColoredPoint.java`.

5.5 Setters

As Java, Jam has several assignment operators other than =; clearly, each of them generates different setters. For all operators, except for = and +=, both operands must have a primitive type. If the left operand of += is of type string (`java.lang.String`), then the right operand can be of any type (either primitive or reference). As a consequence, when a field type is primitive or string, more than one setter must be defined.

Depending on the field type, we can distinguish three different cases:

—boolean type;

—nonboolean primitive type;

—string type (`java.lang.String`).

In the first case we need to define a different setter for each of the operators =, |=, &=, and ^=. All these setters have a boolean argument and return a boolean.

In the second case we need to define a setter for every assignment operator; the return type coincides with the corresponding field type, but the argument

```
mixin Shape {

   public String tooltip ;
   public boolean isVisible ;
   public int zOrder ;

}
```

Fig. 52. Declaration of mixin Shape.

must be of type double since the right operand can be of any primitive type except for boolean.

Finally, in the case the field is a string, we need a setter for the operator = and various setters for the operator +=. In particular, we need one setter for every primitive type and one setter for the type java.lang.Object.

Figures 52–54 cover all these cases by showing how the translation works for a class Shape containing three fields of type boolean, integer, and string, respectively.

5.6 Invisible Jacc

The syntactic analyzer has been developed using *Invisible Jacc* (available at http://www.invisiblesoft.com/jacc.html). Invisible Jacc is a parser generator implemented in Java which produces lexical analyzers and parsers in Java. The lexical analyzer reads a source file and produces a stream of tokens (defined by regular expressions); the parser reads tokens and generates terms according to the productions used for defining the parser itself.

Invisible Jacc supports LALR(1), LR(1), and optimized LR(1) grammars. For Jam it has been used an LALR(1) grammar that can be downloaded at http://www.disi.unige.it/person/LagorioG/jam.

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.

6.1 Object-Oriented Languages Supporting Mixins

To our knowledge, the only existing proposals for extensions of object-oriented languages with mixins are Bracha and Griswold [1996], Flatt et al. [1998], and Bono et al. [1999].

In Bracha and Griswold [1996] 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

```
//
// File automagically created from "Shape.jam" — DO NOT EDIT !!!
//
// (Jam may overwrite this file next time you'll run it)
//

interface Parent$Shape {

   // Methods:

}

public interface Shape extends Parent$Shape {

   // Field "tooltip"
   java.lang.String Shape_$get$_tooltip() ;
   java.lang.String Shape_$set$_tooltip(java.lang.String newValue) ;
   java.lang.String Shape_$addAndSet$_tooltip(java.lang.Object rightOp) ;
   java.lang.String Shape_$addAndSet$_tooltip(boolean rightOp) ;
   java.lang.String Shape_$addAndSet$_tooltip(byte rightOp) ;
   java.lang.String Shape_$addAndSet$_tooltip(short rightOp) ;
   java.lang.String Shape_$addAndSet$_tooltip(char rightOp) ;
   java.lang.String Shape_$addAndSet$_tooltip(int rightOp) ;
   java.lang.String Shape_$addAndSet$_tooltip(long rightOp) ;
   java.lang.String Shape_$addAndSet$_tooltip(float rightOp) ;
   java.lang.String Shape_$addAndSet$_tooltip(double rightOp) ;

   // Field "isVisible"
   boolean Shape_$get$_isVisible() ;
   boolean Shape_$set$_isVisible(boolean newValue) ;
   boolean Shape_$andAndSet$_isVisible(boolean rightOp) ;
   boolean Shape_$exclusiveOrAndSet$_isVisible(boolean rightOp) ;
   boolean Shape_$inclusiveOrAndSet$_isVisible(boolean rightOp) ;
```

Fig. 53. File Shape.java (first part).

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, and 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. Another difference with respect to our approach (see Section 2.3) is that 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

```
    // Field "zOrder"
    int Shape_$get$_zOrder() ;
    int Shape_$set$_zOrder(int newValue) ;
    int Shape_$multiplyAndSet$_zOrder(double rightOp) ;
    int Shape_$divideAndSet$_zOrder(double rightOp) ;
    int Shape_$remainderAndSet$_zOrder(double rightOp) ;
    int Shape_$addAndSet$_zOrder(double rightOp) ;
    int Shape_$subtractAndSet$_zOrder(double rightOp) ;
    int Shape_$leftShiftAndSet$_zOrder(double rightOp) ;
    int Shape_$signedRightShiftAndSet$_zOrder(double rightOp) ;
    int Shape_$unsignedRightShiftAndSet$_zOrder(double rightOp) ;
    int Shape_$andAndSet$_zOrder(double rightOp) ;
    int Shape_$exclusiveOrAndSet$_zOrder(double rightOp) ;
    int Shape_$inclusiveOrAndSet$_zOrder(double rightOp) ;


    // Methods:

}
```

Fig. 54. File `Shape.java` (second part).

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 Bracha and Griswold [1996], 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 a matter of further work, even though a generalization allowing full mixin composition seems in the Java case not trivial, on both the design and implementation sides.

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

In Flatt et al. [1998] 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 methods the expected superclass should provide, that is, the methods which are specified as `inherited` in Jam.

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 (see below).
—*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 Flatt et al. [1998] 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 allows unexpected overriding, while in Jam this is forbidden. This different policy is probably the most important difference between the two approaches. We have already discussed the two opposite requirements of keeping type system simple or keeping run-time semantics simple at the end of Section 2.3. A disadvantage of our approach is that a mixin becomes useless when a parent class incidentally has some method which is in conflict with one defined in the mixin. However, the conflicts resolution in Flatt et al. [1998], essentially based on the idea of keeping both method versions, as happens in multiple inheritance (a class inherits two different definitions for the same method), implies a run-time semantics which significantly deviates from Java semantics (references to objects are heavily complicates both language semantics and a possible implementation (only outlined in Flatt et al. [1998])), while our choice implies minimal changes with respect to Java semantics. A future development could be the analysis of intermediate solutions.

In Bono et al. [1999] an imperative typed calculus for classes and mixins is presented. The language of the calculus is class-based and methods are defined on top of a simple functional language with references and no polymorphic types.

A mixin is defined by a collection of method definitions and a constructor. There are two kinds of method definitions: methods declared *new* cannot be inherited by the parent class (that is, no interference is allowed), whereas those declared *redefined* override those inherited from the parent class. Note that a redefined method must be provided by the parent class; therefore it corresponds to a Jam method which is both defined and declared inherited. Except for redefined methods, inherited methods are not explicitly declared in the mixin but are inferred from method bodies together with their types. Methods (either new

or redefined) can be protected (hence, they are visible to subclasses but not to clients) but not overloaded.

Classes, except for the predefined root class *Object*, are only created by applying a mixin to a class.

A mixin type specifies the names and the types of inherited methods (including redefined methods and those inferred from method bodies) and of defined methods (both redefined and new).

An operational semantics and a type system (proved to be sound) are defined.

Both the static and dynamic semantics of mixin application are similar to those of Jam; in particular, the parent class cannot contain methods declared new in the mixin (so that interference is forbidden) and must provide all inherited methods of the mixin type. Extra methods which do not interfere are allowed and are visible in the resulting class obtained by application. Finally, since method overloading is forbidden, the type of redefined methods can be specialized as well as the type of the methods provided by the parent class and declared inherited in the mixin type.

The main differences with our work are the following:

—the proposed language is theoretical, like MixedJava;
—the type system is based on type inference, rather than type checking;
—inheritance and object subtyping are separate; mixins and classes are not object types, whereas both width and depth structural subtyping is allowed between objects;
—classes and mixins are first class values;
—the absence of method overloading simplifies a lot the type system.

6.2 Mixins versus 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, for example; Pizza [Odersky and Wadler 1997], its evolution GJ [Bracha et al. 1998], and PolyJ [Meyers et al. 1997]; 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 Section 2.6.

Hence, there are cases where heir classes cannot be "abstracted" in a mixin definition. As shown in Section 2.6, this could be achieved by introducing canonical notation for the parametric names of the parent and heir class:

```
mixin M {
  public H* m() {return this;}
  public P* m() {return this;}
}
```

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, mixin names 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.

6.3 Mixins versus Modules

A different direction for extending the expressiveness of standard object-oriented languages, Java in particular, is to add to the language a module level. This kind of extension, differently from both the extensions with mixins as Jam and those discussed in Section 6.1, and the extensions with parametric types discussed in Section 6.2, preserve the "core" language as it stands and only add a new language layer where the programmer can declare modules whose components are Java classes, and combine them. In this way it is possible to simulate a variety of features, including generic types and mixin classes [Ancona and Zucca 2001] and even "upside-down" mixins [McDirmid et al. 2001]. Proposals in this direction are, to our knowledge, the simple extension of the package system proposed in Bauer et al. [1999] and the JAVAMOD language presented in Ancona and Zucca [2001], which is a true module language in the spirit of, for example, ML modules, hence supporting module types and module combinators, and where module components are Java classes, and Jiazzi [McDirmid et al. 2001], which is not actually an extension at the language level but rather a system that enables the construction of large-scale binary components in Java. Note that extensions with modules do not allow a simple translation in Java code as it is for Jam since, roughly, they allow different views of the same class inside different modules and this cannot be simulated in standard Java (see Ancona and Zucca [2001] for more on this point). Indeed, what happens, for instance, in the Jiazzi system is that composition (linking) of two modules is an operation at the bytecode level; see Section 5.1 for more comments on the implementation side.

Even though languages like JAVAMOD and Jiazzi offer more powerful mechanisms for combining code, their type systems are not so flexible as one could expect; for instance, neither JAVAMOD nor Jiazzi provide the Jam notion of mixin type which allows users to partly recover multiple inheritance; for this reason, we can state that the expressive power of either JAVAMOD or Jiazzi is not comparable with that of Jam.

6.4 Alternative Design Choices

6.4.1 *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`.

Further enhancements could also include a means of renaming methods to make them match an `inherited` specification.

6.4.2 *Shared Static Components.* In Section 2.2.4, we have seen that each mixin instance has its own copy of the static components declared in the mixin. As already mentioned there, two other design choices would be possible: either to make mixin instances to share a unique copy of each static component (in this way they would be part of the mixin type), or to leave to the user, by means of a keyword `shared` or an 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

ANCONA, D., LAGORIO, G., AND ZUCCA, E. 2000. Jam: A smooth extension of Java with mixins. In *ECOOP'00—European Conference on Object-Oriented Programming*, E. Bertino, Ed. Lecture Notes in Computer Science, vol. 1850. Springer, Berlin, Germany, 154–178.

ANCONA, D., LAGORIO, G., AND ZUCCA, E. 2001. A core calculus for Java exceptions. In *ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications 2001*. ACM Press, New York, NY.

ANCONA, D. AND ZUCCA, E. 1998. A theory of mixin modules: Basic and derived operators. *Math. Struct. Comput. Sci. 8*, 4 (Aug.), 401–446.

ANCONA, D. AND ZUCCA, E. 2001. True modules for Java-like languages. In *ECOOP'01—European Conference on Object-Oriented Programming*, J. Knudsen, Ed. Lecture Notes in Computer Science, vol. 2072. Springer, Berlin, Germany, 354–380.

ANCONA, D. AND ZUCCA, E. 2002. A calculus of module systems. *J. Funct. Program. 12*, 2 (March), 91–132.

BANAVAR, G. AND LINDSTROM, G. 1996. An application framework for module composition tools. In *ECOOP'96—European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, vol. 1098. Springer, Berlin, Germany, 91–113.

BAUER, L., APPEL, A., AND FELTEN, E. 1999. Mechanisms for secure modular programming in Java. Tech. Rep. CS-TR-603-99, Department of Computer Science, Princeton University, Princeton, NJ.

BONO, V., PATEL, A., AND SHMATIKOV, V. 1999. A core calculus of classes and mixins. In *ECOOP'00—European Conference on Object-Oriented Programming*, R. Guerraoui, Ed. Lecture Notes in Computer Science, vol. 1628. Berlin, Germany, 43–66.

BOOCH, G., RUMBAUGH, J., AND JACOBSON, J. 1998. *Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA.

BRACHA, G. 1992. The programming language JIGSAW: Mixins, modularity and multiple inheritance. Ph.D. thesis, Department of Computer Science, University of Utah, Salt Lake City, Utah.

BRACHA, G. AND COOK, W. 1990. Mixin-based inheritance. In *ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications 1990*. *SIGPLAN Not. 25*, 10, 303–311.

BRACHA, G. AND GRISWOLD, D. 1996. Extending Smalltalk with mixins. In *OOPSLA96 Workshop on Extending the Smalltalk Language*. Electronic note available online at `http://www.javasoft.com/people/gbracha/mwp.html`.

BRACHA, G. AND LINDSTROM, G. 1992. Modularity meets inheritance. In *Proceedings of the International Conference on Computer Languages*. IEEE Computer Society, Press, Los Alamitos, CA, 282–290.

BRACHA, G., ODERSKY, M., STOUTMIRE, D., AND WADLER, P. 1998. Making the future safe for the past: Adding genericity to the Java programming language. In *ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications 1998*. Available online at `http://www.cs.bell-labs.com/who/wadler/pizza/gj/`.

DROSSOPOULOU, S. AND EISENBACH, S. 1999. Describing the semantics of Java and proving type soundness. In *Formal Syntax and Semantics of Java*, J. Alves-Foss, Ed. Lecture Notes in Computer Science, vol. 1523. Springer, Berlin, Germany, 41–82.

DROSSOPOULOU, S., EISENBACH, S., AND WRAGG, D. 1999. A fragment calculus—towards a model of separate compilation, linking and binary compatibility. In *Proceedings of the IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, CA.

DUGGAN, D. AND SOURELIS, C. 1996. Mixin modules. In *Proceedings of the International Conference on Functional Programming*. ACM Press, New York, NY, 262–273.

FISHER, K. AND REPPY, J. 1999. The design of a class mechanism for MOBY. In *PLDI'99—ACM Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, 37–49.

FLATT, M., KRISHNAMURTHI, S., AND FELLEISEN, M. 1998. Classes and mixins. In *ACM Symposium on Principles of Programming Languages 1998*. ACM Press, New York, NY, 171–183.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, MA.

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2000. *The Java™ Language Specification*, 2nd ed. Addison-Wesley, Reading, MA.

KEENE, S. 1989. *Object Oriented Programming in Common Lisp: A Programming Guide in CLOS*. Addison-Wesley, Reading, MA.

MCDIRMID, S., FLATT, M., AND HSIEH, W. 2001. Jiazzi: New age components for old fashioned Java. In *ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications 2001*. ACM Press, New York, NY.

MEYERS, A., BANK, J., AND LISKOV, B. 1997. Parameterized types for Java. In *ACM Symposium on Principles of Programming Languages 1997*. ACM Press, New York, NY.

MOON, D. 1986. Object oriented programming with flavors. In *ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications 1986*. ACM Press, New York, NY, 1–8.

ODERSKY, M. AND WADLER, P. 1997. Pizza into Java: Translating theory into practice. In *ACM Symposium on Principles of Programming Languages 1997*. ACM Press, New York, NY.

SNYDER, A. 1986. CommonObjects: An overview. *SIGPLAN Not. 2*, 10, 19–28.

Received January 2002; revised April 2002; accepted February 2003