Smart Modules for Java-like Languages[*]
(Extended abstract)

Davide Ancona, Giovanni Lagorio, and Elena Zucca

DISI - Università di Genova
Via Dodecaneso, 35, 16146 Genova (Italy)
email: {davide,lagorio,zucca}@disi.unige.it

Abstract. We present SMARTJAVAMOD, a language of mixin modules supporting *compositional compilation*, and constructed on top of the Java language. More in detail, this means that basic modules are collections of Java classes which can be typechecked in isolation, inferring constraints on missing classes and allowing safe reuse of the module in as many contexts as possible. Furthermore, it is possible to write structured module expressions by means of a set of module operators, and a type system at the module level ensures type safety, in the sense that we can always reduce a module expression to a well-formed collection of Java classes. What we obtain is a module language which is extremely flexible and allows the encoding (without any need of enriching the core level, that is, the Java language) of a variety of constructs supporting software reuse and extensibility.

1   Introduction

One of the main reasons of the success of the object-oriented paradigm is its good support for software reuse and extensibility; yet, mainstream object-oriented strongly typed languages still suffer from problems which prevent code to be easily and modularly reused in many circumstances.

Witness of this problem is the amount of recent papers which can be found in literature which aim at enhancing software extensibility in object-oriented languages[12, 3, 2, 14, 13, 8]. The proposed solutions are very disparate, ranging from the introduction of single new abstraction mechanisms that can be more or less easily integrated with existing languages, to more general and pervasive solutions leading to new languages and even paradigms [25, 21].

The approach followed here is partly different from both, and is based on the idea that software extensibility can be enhanced without extending a language in itself, but rather by building on top of this language a *module system*, as it has been successfully achieved in the functional paradigm (see SML for instance). Indeed, Java (and the other mainstream object-oriented languages) lack a real

module system, partly because the notion of class is overloaded and, in fact, classes play the double role of object templates and modules. However, as firstly recognized by Szyperski [28], classes do not offer all features expected from a modern module system (see also [16] for an extended discussion). Java packages are a symptom of this deficiency: they provide a better support than classes for name space management and code structuring, but still they are not real modules.

In particular, the module system we consider in this paper is based on the notion of *mixin module*, which has received a quite consolidated consensus as an effective means for enhancing software reuse. Mixin modules are a generalization, due to Cook and Bracha [10, 11], of the notion of *mixin class* (heir class parametric in the parent) [9, 18, 2]. With respect to conventional modules, mixin modules offer advanced features including *mutually recursive* modules and *virtual* module components which can be redefined via an *overriding* operator. Mixin modules are equipped with a quite powerful set of operations which allow flexible manipulation and reuse of code in a type safe style, and they offer general and uniform solutions to the problem of software extension, so that some other mechanisms (as, for instance, traits, generic classes, mixin classes) can be, totally or partially, recovered in them.

While the theoretical foundations of mixin modules have been already investigated in details [4], much work still needs to be done for integrating real languages with systems of mixin modules. In particular, designing a system of mixin modules on top of Java is not a simple task. One of the main challenges is to devise a type system expressive enough for supporting separate typechecking and compilation of modules, without losing the potentialities offered by mixins for reusing and extending code. In [3], we described a proposal, called JAVAMOD (supporting separate typechecking via module interfaces, and module expressions constructed by *merge*, *renaming* and *hiding* operators), which, however, had many limitations. In particular, the programmer had to explicitly annotate with their required types the classes used as parameters in modules, thus making module declarations cumbersome; the type system was too restrictive (see below); furthermore, virtual classes were not supported.

In this paper we present SMARTJAVAMOD, a major revision of the design given in [3], with the aim of defining a much more expressive type system, and a richer set of module operators supporting virtual classes and, more generally, code reuse in presence of unanticipated software modifications.

The key novelties of the approach are the following:

– Type requirements on classes used in a module are *automatically inferred*, without any need for extra type annotations provided by the programmer. We call *smart modules* our modules to stress that, thanks to the type inference algorithm, a basic module (that is, a named collection of class declarations possibly parametric in some classes) can be typechecked and compiled in *full isolation*, without any need that either used classes in source/bytecode form or even a specification file is available. Moreover, these type requirements are *the minimal requirements* on the context needed to safely use a

module. This can be achieved by using the recent technology developed in [1], where a type system supporting *principal typings* [**?**] for Featherweight Java [20] has been defined. More in detail, in [1] the type information inferred for a class c is a pair consisting of the set of constraints (possibly involving type variables) on all classes used in the code of c (c included) sufficient and necessary for successfully compiling c, and of the type of c which can be easily extracted from its declaration by removing all the method bodies. Furthermore, the notion of *polymorphic bytecode* (bytecode where type annotations can contain type variables) supports separate bytecode generation for classes.

– Classes declared in a module are all implicitly *virtual*, that is, their definition can be later replaced when composing the module with others. The standard way to override components (classes in this case) of mixin modules is by means of a *restrict* operator [10, 4] which allows the deletion of a component definition, in such a way that a new definition can then be provided by merging the module with another. In order to support this mechanism, here type inference uses the same algorithm proposed in [1], but, differently from there (where constraints were incrementally simplified when linking classes together), simplification of constraints is never performed, since they clearly need to be re-checked in case some class declaration is overridden.

– Finally, in addition to classical operators for manipulating mixin modules [10, 4], such as *merge* and *renaming* (already present in [3]), and *restrict*, we introduce here the novel *bind* and *unbind* operators; the former is used to associate module parameters with class declarations already present in the same module, the latter to introduce a new module parameter by removing an existing association. These operators can be expressed in terms of primitive module operators (namely, by using *reduct* [4]), but to our knowledge have never been proposed as high-level constructs in previous languages supporting mixin modules. In this paper, we introduce them (in particular *unbind*) as a powerful means, yet with simple and clean semantics, for supporting unanticipated software evolution, in what it allows an external user to take some code which was designed as closed and to make it parametric, hence available to further composition.

The formalization of the module system is carried out with Featherweight Java as the underlying language. The semantics of module expressions is defined by a set of rewriting rules which allow simplification of each module expression into a basic module. The type system turns out to be sound with respect to the rewriting semantics: the usual subject reduction and progress properties are stated.

The paper is organized as follows. In Section 2 we discuss more in detail the key ideas at the basis of smart modules. Section 3 is a gentle introduction to SMARTJAVAMOD, while Section 4 shows the expressive power of the language and its ability to support software reuse and unanticipated software evolution by using a complete running example. The formalization of the semantics and type system of smart modules for Featherweight Java can be found in Section 5, while

3

Section 6 gives some hints towards a possible implementation. Finally, Section 7 contains pointers to related work and some final remarks and directions for future developments.

2   Smart Modules

Java and other widely used object-oriented languages, like C++, provide only weak module systems. Java packages and C++ namespaces, for instance, are just a means for structuring the namespace, and Java/C++ classes incorporate some, but not all, features found in richer module mechanisms.

For instance, visibility levels are handled at the class level, while a separation of concerns allows a more clean and powerful language design, as already pointed out in [28] (see also [16] for an extended discussion on this point).

Despite the considerable effort invested in studying theoretical foundations and developing new forms of module systems[1], there is relatively little amount of effort, both on the theoretical and implementation side, on applying these ideas to the case of object-oriented languages.

Highly desirable features for a modular approach to software development are:

Separate type-checking A code fragment (module) which needs some services from others should be possibly typed in isolation. In Java the type-checking of a class which relies on other classes requires all the *definitions* of these classes in the environment. For instance, consider the following simple class, representing single-linked lists (where N, not present, is the type of a node):

```
class List {
  N first;
  void addFirst() { first=new N(first); }
}
```

This class cannot be compiled if the definition of class N is unavailable. One reason is the fact that a Java compiler, in order to typecheck the expression `new N(first)`, needs to
  - know which (accessible) constructors class N provides,
  - choose the most specific among the applicable ones[2],
  - annotate the generated bytecode for the invocation with the parameter types of the chosen constructor.

For instance, the compiled bytecode might look, from a very abstract point of view, like `new N<<Object>>(first)` if class N provides a constructor which receives an `Object` (and does not provide a more specific constructor).

---

[1] Let us mention the wide literature about foundations and improvements of Standard ML module system (see, e.g., [22, 19]), the notions of *mixin* (see, e.g., [11, 10, 4]) and *unit* [17] and the type-theoretical analysis of recursion between modules proposed in [15].

[2] If a most specific constructor cannot be found, then an error is raised and the compilation aborted.

As the name `N` may be seen as a parameter for class `List` and generics[3] allow to abstract over types, one might be tempted to transform `List` into `List<N>` in order to compile its definition in isolation:

```
class List<N> {
  N first;
  void addFirst() {
      first=new N(first) ; // ERROR
  }
}
```

Unfortunately, this is not possible for various reasons: first of all, `N` can be used only as a type, that is, instances of `N` cannot be created using `new` operator. Also, inside the body of `List<N>` all you can do with an object of type `N` is determined by its upper bound[4]; this strongly limits the reuse of a module, as we discuss further below. On the contrary, a similar solution works with C++ templates, just because in that case there are no limitations on the use of `N`. However, since typechecking and compilation of templates is postponed at instantiation time, the C++ solution does not meet our modularity requirements.

From an implementation point of view, generics are currently implemented by the Java compiler using type erasure so, all instances of `List<N>` are translated to the same class `List` and, roughly speaking, the identity of `N` is unknown at runtime. Anyway, even if the identity of `N` were not lost at runtime (some extensions of Java, as *LOOJ* [13] and *NextGEN* [27], do not lose it) the fact that in Java constructors are not inherited makes impossible to know which constructors are available given only the upper bound of a type parameter.

Module interfaces A module interface is a specification of the services a given module both needs from and provides to others. As it is well-known, these specifications serve as a formal contract [26] between server and client. Hence, the client can rely on the server specification without any need at looking at the implementation; on the other hand, the correctness of an implementation w.r.t. the specification can be separately checked.

Module expressions A principle which has recently become popular in the programming language community (see, e.g., [23]) is that a module system should have two linguistic levels: a *module language* providing operators for combining software components, constructed on top of a *core language* (following the terminology introduced with Standard ML) for defining module components. The module language should have its own typing rules and be as independent as possible from the core language; even more, it could be in principle instantiated over different core languages.

Module interfaces and separate type-checking usually go hand in hand as the former is used to implement the latter. For instance, *Jiazzi* [24], *Keris* [30] and

---

[3] Introduced in JDK 1.5.

[4] In this example the upper bound is implicitly the default one: class `Object`.

our JAVAMOD [3] are three proposals which build a module system upon Java, enjoy the desirable features listed before, and require the user to write module interfaces. This last requirement, that is, requiring *the user* to write an explicit module interface is the source of two drawbacks. The more obvious is that the user is forced to write more than she/he would probably like to; the other, less obvious, drawback is the fact that fixing an interface, expressed as a sequence of fields/methods and so on, severely limits the reuse of a module because such an interface, in most cases, is more specific than it needs to be. Let us explain both points by means of an example; consider again the module for defining single-linked lists, sketched before. In JavaMod it can be written as the following basic module:

```
interface ParametricList is
N: { N(N) } ->
   List : { N first ; void addFirst() ; }

module ParametricList is
  class List {
     N first;
     void addFirst() { first=new N(first); }
  }
```

The definition of a basic module is split in two parts: a module interface which describes what a module expects and provides, and a module definition. In this example the module interface describes that, given a (parameter) class N which provides a constructor which receives, as the only argument, an object of the same class, the module ParametricList provides a class named List with an N field called first and a void method called addFirst.

Let us consider the following module:

```
interface Node is
-> N : { N(Object) }

module Node is
  class N {
     N(Object) {}
  }
```

When two modules are merged, the expected and provided interfaces are matched for equality, so the module Node cannot be merged with module ParametricList because class N does not provide a constructor which receives an N.

Yet, class List can be successfully typechecked and compiled in an environment where the only constructor for N receives an Object; one definition of N is as good as the other. So, from the point of view of module ParametricList, the most general requirement on the parameter N is not simply "to provide a constructor which receives an N", as expressed by a classic module interface, but "to provide a constructor which can be unambiguously called passing an N".

Keeping the module interfaces unaltered and relaxing the check on interface matching is not an option because, as we recalled before, in order to compile the expression `new N(first)`, the compiler must know *exactly* which is the type of the parameter of the chosen constructor. Moreover, relaxing the requirements on interface matching and taking into account overloading resolution would be a very challenging task.

It may appear that trying to make the module interfaces more precise, in order to capture more reuse contexts, would put more burden on the programmer.

Interestingly, we can disburden the programmer from writing module interfaces *and* retain all the desirable features listed before in one fell swoop using inferred type constraints and polymorphic bytecode [1].

As the rest of the paper illustrates, in the new approach we propose here, the programmer does not need to write explicitly the interfaces of the modules a basic module `M` depends on, as the most general requirements needed for a successful compilation of `M` are automatically inferred by the compiler, which can compile `M` in *total isolation*.

3 Language Overview

This section is a brief introduction to smart modules: their main features are presented through some simple, but still meaningful, examples showing their expressive power.

For convenience, the underlying core language used in the examples of this section is Java, but all code could be easily rewritten in C#; however, for simplicity, the formal treatment given in Section 5 will consider only a subset of Java, namely Featherweight Java [20].

3.1 Basic Smart Modules

Let us start our introduction with an example[5] of declaration of basic module:

```
module LinkedList{ // parametric in N
 class List{
  N first;
  void addFirst(){first=new N(first);}
 }
 class Node{
  N next;
  Node(N n){next=n;}
  N getNext(){return next;}
 }
}
```

A basic module is a collection of class/interface declarations (like `List` and `Node`) possibly parametric in classes/interfaces (like `N`) which can be bound to

_____
[5] For simplicity, we will keep the examples small and avoid access modifiers.

their definition later. In this example the name `N` is a parameter, as no types named `N` are declared inside the module.

Within this example, the intuition is that `N` *could be* `Node`; indeed, if we replaced all occurrences of `N` with `Node`, then we would obtain the classic example of single-linked lists. Of course, having used a parameter instead of the name of a fixed class, allows us to bind `N` to something more specific than `Node` later, for instance a class `DoubleNode` (which, presumably, extends `Node`).

This particular use of a module parameter roughly corresponds to the idea of type `ThisClass` of LOOJ [13].

Our approach allows a step further though: `N` can be bound to *any class* that provides an accessible constructor which can be invoked passing an object of type `N` as the single argument; that is, a constructor which is declared to receive a `N` or any of its supertypes (superclasses or implemented interfaces, whatever they might be).

Note that even if the programmer does not have to declare:

1. which are the parameters of a module (in this case, `N` is the only one) and
2. what features (methods, fields and so on) `N` is supposed to provide,

our approach allows to compile this basic module in *total isolation* (no other sources or binary `.class` files are needed) and to compose this module with others in a *type safe* manner.

The point is that we do not require the programmer to write the interfaces of used modules because we can *infer* the most general requirements on used modules, which can be later used to check whether module compositions are type safe or not.

For instance, the type inferred for `LinkedList` is

```
{
 List:{cons(N,N,'a)} ->
     extends Object {N first; void addFirst();}
 Node:{exists(N)} ->
     extends Object {N next; Node(N n); N getNext();}
}
```

The type of a module is a mapping from the class names declared in the module to their corresponding class types. In a class type, the set on the left hand side of `->` contains all the type constraints needed for successfully compiling the class, while the right hand side provides all the type information on the class. For instance, the type of `List` means that the class can be successfully compiled if and only if `N` is bound to a class declaration providing a constructor which is the most specific for an argument of type `N` (hence, having a parameter of type `'a`, with `'a` a type variable such that `N <= 'a`). Similar constraints are used for dealing with field access and method invocations.

Note that the type variables which occur in the constraints are all existentially quantified and their scope is limited to the left hand side of `->`; more precisely, the type of `List` should be rewritten as follows:

```
( ∃ 'a.{cons(N,N,'a)}) ->
    extends Object {N first; void addFirst();}
```

The fact that the right hand side of `->` might contain type variables as well merely depends on the underlying programming language: for instance, only the latest version of Java (JDK 1.5) allows generic classes and polymorphic methods.

Since `N` is used in `Node` as a type, the only constraint is `exists(N)` which requires class `N` to be bound to an existing class declaration.[6]

3.2 Open and Closed Modules

Classes `List` and `Node` inside `LinkedList` cannot be used until the parameter `N` is bound to a class declaration which satisfies the constraints of `List` and `Node`.

```
// the following is a type error,
// since N has not been instantiated yet

LinkedList.List l=new LinkedList.List(); // ERROR
```

A module with parameters, such as `LinkedList`, is called *open*. The module system offers two different operators for instantiating parameters of open modules: *bind* and *merge*.

Bind The bind operator allows self-instantiation. For instance, since class `Node` satisfies all required constraints on `N`, we can define a module `ClosedLinkedList` obtained from `LinkedList` by binding `N` to `Node`:

```
module ClosedLinkedList=bind(LinkedList,{N->Node}) ;
```

Module `ClosedLinkedList` is *closed*, since it has no parameters. Equivalently, `ClosedLinkedList` could have been declared by copying the definition of `LinkedList` and replacing each occurrence of `N` with `Node`.

```
module ClosedLinkedList { // naive cut-and-paste approach

 class List {
  Node first;
  void addFirst() {first=new Node(first);}
 }
 class Node {
  Node next;
  Node(Node n) {next=n;}
  Node getNext() {return next;}
 }

}
```

Now classes `List` and `Node` can be used:

----

[6] Note that `cons(N,N,'a)` subsumes `exists(N)` and `N<='a`.

```
ClosedLinkedList.List l=new ClosedLinkedList.List();
```

When closing a module, all type constraints in the class types must be verified, otherwise a type error is issued.[7] For instance, the expression `bind(LinkedList,{N->List})` is not type correct, since `List` does not satisfy the constraint `cons(List,List,'a)`.

The reader might expect that the type of a closed module contains no constraints, but this is not the case. Indeed, a closed module is not "sealed" once and for all, but can be reopened using operators *restrict* and *unbind*, which we discuss in Section 3.4. So, for instance, the type of `ClosedLinkedList` is

```
{
 List:{cons(Node,Node,'a)} ->
    extends Object {Node first; void addFirst();}
 Node:{exists(Node)} ->
    extends Object {Node next; Node(Node n); Node getNext();}
}
```

Finally, note that the fact that a module as `ClosedLinkedList` is closed can be simply deduced from its type: it is immediate to see that all class names contained in the type of `ClosedLinkedList` are declared inside the module itself.

Merge Assume we want to extend the code in `LinkedList` in order to support doubly linked lists. This extension can be isolated in a separate module:

```
module Double { // three parameters: N, List and Node
 class DoubleList extends List {
  N last;
  void addLast() {
    N n = new N(last, null) ;
    if (first==null) first = n ;
    if (last!=null) last.next = n ;
    last = n;
  }
  void addFirst() {
    N n=new N(null, first);
    if (first!=null) first.prev = n ;
    first = n ;
    if (last==null) last=n ;
  }
 }
 class DoubleNode extends Node {
  N prev;
  DoubleNode(N n) {super(n);}
  DoubleNode(N p,N n) {super(n); prev=p;}
  N getPrev() {
   return prev;
  }
 }
}
```

---

[7] For constraint verification see also Section 3.5.

Now we can define `DoubleLinkedList` by merging `LinkedList` with the extension defined in `Double`:

```
module DoubleLinkedList=merge(LinkedList,Double);
```

Note that the two parameters `List` and `Node` of module `Double` have been automatically instantiated in `DoubleLinkedList` with the corresponding classes declared in `LinkedList`, whereas the parameter `N` is shared (instantiation and sharing of parameters are by class name matching).

Now we can instantiate `N` with `DoubleNode`:

```
module ClosedDoubleLinkedList =
    bind(DoubleLinkedList, {N->DoubleNode}) ;

// this below would be a type error, since
// DoubleList requires N to satisfy
// cons(N,(N,Null),('a,'b))
//
// module ClosedDoubleLinkedList =
//   bind(DoubleLinkedList, {N->Node}) ; // ERROR
```

3.3   Renaming Facilities

Assume that, instead of `N`, `List` and `Node`, the three parameters of module `Double` have been named `DN`, `L` and `N`, respectively. Since, when merging modules, parameter instantiation and sharing is by name matching, we need a renaming operator for correctly defining module `DoubleLinkedList` as above.

```
module DoubleLinkedList =
   merge(LinkedList,
         rename(Double,{N->Node,L->List,DN->N})) ;
```

The *rename* operator allows renaming of a single class name at time, therefore the syntax we have just used is just a convenient shortcut for the more verbose expression:

```
rename(rename(rename(Double,N->Node),L->List),DN->N)
```

This means that renaming of more components is accomplished sequentially from left to right. Both parameters and declared classes can be renamed, however the new name must be unused in order to avoid conflicts; in other words, the operator allows only bijective renaming.

3.4   Virtual Classes and Unanticipated Code Modification

Let us consider again module `ClosedLinkedList` as defined in Section 3.2. As already noted, the type of the module contains constraints on `Node`, even though `Node` is not a parameter. This is because all classes defined in a module are *virtual*, in the sense that their definitions can be overridden; hence, constraints involving a class must be kept in order to be sure that they are still satisfied by

11

later redefinitions of the other classes. Since the merge operator does not allow class overriding (trying to merge modules which declare classes with the same name is a type error), class redefinition can be accomplished in two steps by means of the *restrict* operator.

For instance, let us assume we want to define a new module `ClosedIntLinkedList` obtained from `ClosedLinkedList` by overriding the definition of `Node`:

```
module IntNode {
 class Node {
  Node next;
  int elem;
  Node(Node n) {next=n;}
  Node(Node n,int e) {next=n;elem=e;}
  Node getNext() {return next;}
  int getElem() {return elem;}
 }
}
module ClosedIntLinkedList =
    merge(restrict(ClosedLinkedList, Node), IntNode);
```

First, we remove the declaration of `Node` in `ClosedLinkedList` by means of the restrict operator. Then we add the new declaration contained in `IntNode` with the merge operator.

Now it should be clear why type constraints must always be kept: if we had removed the constraint `cons(Node,Node,'a)` from the type of `ClosedLinkedList`, we would not be able to correctly typecheck the definition of `ClosedIntLinkedList`.

In general, the restrict operator allows the removal of class declarations in a module (trying to remove classes which are not declared is a type error). As for renaming, a convenient shortcut is provided for allowing restriction of multiple classes (however here the order is immaterial).

A preferential merge operator resolving conflicts between class declarations can be obtained as a more powerful form of syntactic shortcut:

```
merge(M1 < M2)
```

is an abbreviation for

```
merge(restrict(M1,{C1,..,Cn}),M2)
```

where `C1,..,Cn` are all the classes declared in both modules.

Finally, *unbind* is another operator which, like restrict, enhances code reuse in the presence of unanticipated software modifications. For instance, the class `Node` in module `ClosedLinkedList` as defined in Section 3.2 cannot be directly reused for defining doubly linked nodes as it can be done with the class `Node` in module `LinkedList` declared in Section 3.1. However, the availability of `LinkedList` is not essential, since that module can be obtained from `ClosedLinkedList` by unbinding `Node`.

```
module LinkedList=unbind(ClosedLinkedList,{Node->N})
```

The unbind operator is the inverse of bind: the class to be unbound (`Node` in the example) must be declared in the module while the new name (`N` in the example) must be unused. The effect consists in replacing all non defining occurrences of `Node` with `N`. A defining occurrence is either that immediately following the `class` keyword, or those introducing constructor definitions.

3.5   More on Typing

In this final part more details on type inference for smart modules can be found. We refer to Section 5 and to the related paper [5, 1] for a complete technical treatment of the topic.

Constraint Verification   Let us consider the declaration of the following open module:

```
module Vain {// parametric in D
 class C extends D {
  C m() {return new D();}
 }
}
```

In order to be able to use class `C`, the parameter `D` needs to be instantiated. However, it turns out that there is no way to correctly bind `D` to a class declaration: in order to do that the constraint `D <= C` in the type of `C` should be satisfied, but class `C` extends `D`, therefore `C < D` holds which is in contradiction with `D <= C`. As a matter of fact, module `Vain` is rather useless. From a software engineering point of view, declarations of open modules, like `Vain`, which cannot be closed should be avoided in favor of earlier error detection.

The type inference algorithm defined in [1] is smart enough to reject the declaration of `Vain`; however, it is not so smart to recognize all possible cases. Luckily, verification of constraints is complete in case of closed modules [1]: typing of a closed module succeeds if and only if all its classes are type compatible.

Dot notation   As in almost all module systems, smart modules support the dot notation for accessing classes declared in other modules. However two main restrictions apply to the use of the dot notation in order to avoid a too complex type system.

1. As already seen in Section 3.2, dot notation is only allowed for closed modules, in order to avoid run-time errors. For instance, referring to module `LinkedList` defined in Section 3.1, if we try to evaluate the ill-typed expression

   ```
   new LinkedList.List().addFirst()
   ```

   then the exception `NoClassDefFoundError` is thrown when trying to invoke the constructor in the body of `addFirst()`.

2. The most general syntax `ME.c`, where `ME` is a generic module expression, is not allowed; instead, the correct syntactic form is `M.c`, where `M` is a module name. This is the same restriction adopted for the module systems of SML and OCaml in order to avoid a too complex definition of type equivalence. Here we would have the same problems, since in Java a class name denotes a type as well.

Generativity versus Transparency Let us consider the following two module declarations:

```
module ClosedLinkedList =bind(LinkedList,{N->Node})
module ClosedLinkedList2=bind(LinkedList,{N->Node})
```

Should the two types `ClosedLinkedList.List` and `ClosedLinkedList2.List` be considered equivalent? This is a well known issue in ML module systems where the following terminology has been adopted: in case the two types above have to be considered unrelated, the two corresponding module declarations are said to be *generative* (or *opaque*), otherwise they are said to be *transparent*.

In Java all class declarations are generative; for instance, the same class declaration in two different packages always denotes two unrelated types. For analogy one can follow the same approach for modules and, indeed, this is our choice here, but mainly because transparency makes the type system more complex. Let us consider, for instance, the following artificial example:

```
module M1 { // parametric in C
 class A {}
 class B extends C {}
}
module M2=merge({class C{int i;}}, M1);
module M3=merge({class C{boolean b;}}, M1);
```

Let us assume a transparent declaration for `M2` and `M3`. Clearly, `M2.C` and `M3.C` can only be considered unrelated types. Concerning the other two classes `A` and `B`, while `M2.A` and `M3.A` can be safely considered equivalent, the same cannot be done for `M2.B` and `M3.B`, even though `B` comes from the same declaration.

Although the generative approach allows a simpler type system, transparency is advocated in some cases. For this reason, transparent smart modules is an issue that deserves future investigation.

4   Smart Modules at Work

In this section we show the expressive power of smart modules by considering as "benchmark" the classical *expression problem* (or *extensibility problem*). For simplicity we use a slightly more complex variation of Torgersen's example [29]; we refer to the same paper for a comprehensive treatment of the expression problem which would be out of scope in this paper.

The approach we take here is the classical data-centered one, which is more intuitive and simpler, but also less suitable for adding new methods. Our goal is to show that even with the data-centered approach, smart modules allows addition of new methods with full reuse of the code, at the cost of introducing a parameter (for other approaches on adding methods to existing classes see also [8, 14]).

What follows is an implementation of a type `Exp` of simple integer expressions built on top of literals and addition. The only available methods are `clone()`, which allows cloning of an expression, and `print()`, which displays the expression on the screen.

```
module Exp { // parametric in E

 interface Exp {
  E clone();
  void print();
 }

 class Lit implements Exp {
   int value;
   Lit(int v) {value=v;}
   E clone() {return new Lit(value);}
   void print() {System.out.print(value);}
 }

 class Add implements Exp {
   E left,right;
   Add(E l,E r) {left=l; right=r;}
   E clone() {return new Add(left,right);}
   void print() {
       left.print();
       System.out.print('+');
       right.print();
   }
 }
}
```

The code of the module is very similar to a standard data-centered implementation one would write in a Java package. The only difference is the use of the parameter `E` where the standard implementation would use `Exp` instead. As we will see, this is the only complicacy needed for allowing addition of new methods.

The module can be used by a client (for instance a parser) by means of the bind operation.

```
module Client { // parametric in Exp, Lit, Add

 class Producer { // typically, the parser
   Exp produce() {return new Add(new Lit(2),new Lit(3));}
 }
```

```
}
module Application =
      bind(merge(Exp, Client), {E->Exp}) ;
```

For instance, executing the following well-typed statement

```
new Application.Producer().produce().clone().print();
```

will display 2+3 on the screen, as expected.

Assume now we need to extend our code. Since we have chosen a data-centered approach, adding new methods is more challenging than adding new kinds of expressions, therefore let us assume we want to add a new method `eval()` for evaluating expressions.

We can confine the needed extension into a new module:

```
module Eval{ // parameteric in E, Exp, Lit, Add

 interface EvalExp extends Exp {
  int eval();
 }

 class EvalLit extends Lit implements EvalExp {
  EvalLit(int v) {super(v);}
  int eval() {return value;}
 }

 class EvalAdd extends Add implements EvalExp {
  EvalAdd(E l,E r) {super(l,r);}
  int eval() {return left.eval()+right.eval();}
 }

}
```

A first tentative for instantiating the parameters of `Eval` would consist in directly merging `Eval` with `Exp` and, then, binding `E` to `EvalExp`. However, this is not type correct, since, for instance, class `Lit` in `Exp` requires the constraint `Lit <= E` and this would not hold if we replace `E` with `EvalExp`. The real source of the problem is the fact that now the code of `Lit` and `Add` should refer to classes `EvalLit` and `EvalAdd`. This can be accomplished by unbinding `Lit` and `Add` in module `Exp`.

```
module EvalExp =
   merge(Eval, unbind(Exp, {Lit->EvalLit,Add->EvalAdd}));

module Application2 =
   merge(bind(EvalExp, {E->EvalExp}),
         rename(Client, {Exp->EvalExp, Lit->EvalLit,
                         Add->EvalAdd})) ;
```

Now we can execute the following code:

```
Application2.EvalExp e =
  new Application2.Producer().produce().clone();
e.print();
System.out.println(" = "+e.eval());
```

and, as expected, `2+3 = 5` will be displayed on the screen.

Finally, note that, because of generativity, the following code is ill-typed:

```
// this below would be a type error:
// Application2.Exp is not comparable with Application.Exp
Application.Exp e2=e; // ERROR
```

5 Smart Modules for Featherweight Java

The syntax of the core language is defined in Fig.1.

cd ::= class c extends n { fds mds }	$(c \neq$ Object$)$	class declaration
fds ::= n_1 f_1 ; ... n_n f_n ;		field declarations
mds ::= $md_1 \ldots md_n$		method declarations
md ::= mh {return e;}		method declaration
mh ::= n_0 m(n_1 $x_1, \ldots, n_n$ x_n)		method header
e ::= x	e.f	e_0.m($e_1, \ldots, e_n$)	new n($e_1, \ldots, e_n$)	(n)e		expression
n ::= c	M.c		class name

where field, method and parameter names in fds, mds and mh are distinct

Fig. 1. FJ syntax

We consider the same language as in [1], that is, basically Featherweight Java [20] (FJ in the sequel) hence a functional subset of Java with no primitive types, except that here class constructors are implicitly declared, and class names (references to other classes) appearing in code can be, besides simple names, *qualified names* of the form M.c, referring to classes declared in external (closed) modules.

Note that considering a purely functional language here is not a limitation, since modules can only contain classes, and during evaluation of module expressions, classes are not evaluated, hence there would be no side effects even in presence of objects with state.

Every class can contain instance field and method declarations and has only one constructor whose parameters correspond to all class fields (both inherited and declared) in the order of declaration. In class declarations we assume that the name of the class c cannot be Object. Method overloading and field hiding are not supported[8]. Expressions are variables, field access, method invocation,

---

[8] Just for simplicity and for keeping original FJ definition: it is easy to extend the type system in order to support these features.

instance creation and casting; the keyword `this` is considered a special variable.[9] Finally, in order to simplify the presentation, we assume field names in `fds`, method names in `mds`, parameter names in `mh` to be distinct.

The syntax of SMARTJAVAMOD is given in Fig.2.

$\text{MDS} ::= \text{MD}_1 \ldots \text{MD}_n$	module environment
$\text{MD} ::= \texttt{module M is ME}$	module declaration
$\text{ME} ::= \text{M} \mid \texttt{cds} \mid \texttt{merge}(\text{ME}_1, \text{ME}_2) \mid \texttt{restrict}(\text{ME}, \texttt{c}) \mid$	
$\quad\quad \texttt{rename}(\text{ME}, \texttt{c} \rightarrow \texttt{c}') \mid$	module expression
$\quad\quad \texttt{bind}(\text{ME}, \texttt{d} \rightarrow \texttt{c}) \mid \texttt{unbind}(\text{ME}, \texttt{c} \rightarrow \texttt{d})$	
$\texttt{cds} ::= \texttt{cd}_1 \ldots \texttt{cd}_n$	class declarations
where simple class names in `cds` are distinct	

Fig. 2. SMARTJAVAMOD syntax

A basic module is a sequence of class declarations, where for simplicity we assume declared class names to be distinct. A class declaration in a basic module can contain qualified class names, that is, references to classes defined in other (closed) modules, and simple class names. Simple class names can be either *defined*, that is, names referring to classes declared in the module, or *deferred*, that is, names for which a declaration is expected to be provided later when composing the module with others. A module with no deferred class names is said to be *closed*. Note that defined class names are not associated with a class declaration in the module once and for all, but their definition can be changed later when composing the module with others. In other words, module components (classes) are all implicitly *virtual*.[10]

Module operators include merging two modules, removing a class declaration, renaming a class and two operators which allow to bind a deferred class name to a class declaration inside the module and, conversely, to make a class used inside a module deferred.

Reduction rules for module environments and module expressions (in a given module environment) are those given in Fig.3, plus usual contextual closure for module expressions. We denote by $in(\texttt{cds})$ and $out(\texttt{cds})$ the deferred and defined class names in class declarations `cds`, respectively.

We call *output* occurrence of a class name `c` in class declarations `cds` only the occurrence in the declaration `class c extends n { fds mds }`, if any; we call *input* occurrence any other occurrence. We denote by $\texttt{cds}[\texttt{c}'/out\ \texttt{c}]$ the class declarations obtained from `cds` by only replacing the output occurrence of `c`, if

---

[9] We do not consider the keyword `super` even though used in the examples of Section 3. However, it could be added with no cost at the level of type inference.

[10] We consider only virtual classes in this paper for simplicity, since this is enough for the applications we want to illustrate; however, mixin modules typically support also an operator which allows to make a component *frozen* [10, 4], in such a way that references to this component inside the module can no longer be affected by module composition.

$$(mdecs)\frac{\mathtt{MD}_i \longrightarrow_{\mathtt{MDS}} \mathtt{MD}'_i}{\mathtt{MD}_1 \dots \mathtt{MD}_n \longrightarrow_{\mathtt{MDS}} \mathtt{MD}_1 \dots \mathtt{MD}'_i \dots \mathtt{MD}_n} \quad \mathtt{MDS} = \mathtt{MD}_1 \dots \mathtt{MD}_n$$

$$(mdec)\frac{\mathtt{ME} \longrightarrow_{\mathtt{MDS}} \mathtt{ME}'}{\mathtt{module\ M\ is\ ME} \longrightarrow_{\mathtt{MDS}} \mathtt{module\ M\ is\ ME}'}$$

$$(mname)\frac{}{\mathtt{M} \longrightarrow_{\mathtt{MDS}} \mathtt{ME}_i} \quad \begin{array}{l} \mathtt{MDS} = \mathtt{module\ M}_1\ \mathtt{is\ ME}_1 \dots \mathtt{module\ M}_n\ \mathtt{is\ ME}_n \\ \mathtt{M} = \mathtt{M}_i \end{array}$$

$$(merge)\frac{}{\mathtt{merge}(\mathsf{cds}_1, \mathsf{cds}_2) \longrightarrow_{\mathtt{MDS}} \mathsf{cds}_1\ \mathsf{cds}_2} \quad out(\mathsf{cds}_1) \cap out(\mathsf{cds}_2) = \emptyset$$

$$(restrict)\frac{}{\mathtt{restrict}(\mathsf{cd}_1 \dots \mathsf{cd}_n, \mathsf{c}) \longrightarrow_{\mathtt{MDS}} \mathsf{cd}_1 \dots \mathsf{cd}_{i-1}\mathsf{cd}_{i+1} \dots \mathsf{cd}_n} \quad out(\mathsf{cd}_i) = \{\mathsf{c}\}$$

$$(rename)\frac{}{\mathtt{rename}(\mathsf{cds}, \mathsf{c} \to \mathsf{c}') \longrightarrow_{\mathtt{MDS}} \mathsf{cds}[\mathsf{c}'/in\ \mathsf{c}][\mathsf{c}'/out\ \mathsf{c}]} \quad \begin{array}{l} \mathsf{c} \in in(\mathsf{cds}) \cup out(\mathsf{cds}) \\ \mathsf{c}' \notin in(\mathsf{cds}) \cup out(\mathsf{cds}) \end{array}$$

$$(bind)\frac{}{\mathtt{bind}(\mathsf{cds}, \mathsf{d} \to \mathsf{c}) \longrightarrow_{\mathtt{MDS}} \mathsf{cds}[\mathsf{c}/in\ \mathsf{d}]} \quad \begin{array}{l} \mathsf{d} \in in(\mathsf{cds}) \\ \mathsf{c} \in out(\mathsf{cds}) \end{array}$$

$$(unbind)\frac{}{\mathtt{unbind}(\mathsf{cds}, \mathsf{c} \to \mathsf{d}) \longrightarrow_{\mathtt{MDS}} \mathsf{cds}[\mathsf{d}/in\ \mathsf{c}]} \quad \begin{array}{l} \mathsf{c} \in out(\mathsf{cds}) \\ \mathsf{d} \notin in(\mathsf{cds}) \cup out(\mathsf{cds}) \end{array}$$

Fig. 3. Reduction rules for module expressions

any, by c'; we denote by $\mathsf{cds}[c'/in\ c]$ the class declarations obtained from cds by replacing every input occurrence of c by c'. Note that $\mathsf{cds}[c'/out\ c][c'/in\ c]$ corresponds to standard replacement of all occurrences of c by c' in cds, which we will also denote by $\mathsf{cds}[c'/c]$.

Merging two basic modules corresponds to just putting together their class declarations, provided that there are no conflicts (note that deferred class names are shared). The *restrict* operator removes a class declaration, thus making deferred a class name which was defined. The *rename* operator replaces everywhere a class name with a new name. The *bind* operator replaces a deferred class name with the name of a class declared in the module, with the effect of removing a dependency from an external class. Conversely, the *unbind* operator replaces all references to a class declaration inside the module by references to a new name, with the effect of introducing a dependency from an external class.

Types are given in Fig.4.

$$\mathcal{M} ::= (\mathtt{M}_1, \mathtt{MT}_1) \dots (\mathtt{M}_n, \mathtt{MT}_n) \quad \text{module type environment}$$
$$\mathtt{MT} ::= (\varGamma_1, \delta_1) \dots (\varGamma_n, \delta_n) \quad \text{module type}$$
$$\varGamma ::= \gamma_1 \dots \gamma_n \quad \text{constraints}$$
$$\varDelta ::= \delta_1 \dots \delta_n \quad \text{class type environment}$$
$$\delta ::= (\mathsf{c}, \mathsf{n}, \mathsf{fss}, \mathsf{mss}) \quad \text{class signature}$$

Fig. 4. Types

A *module type environment* is a sequence of *module type assignments*, which are pairs consisting of a module name and a module type. A *module type* is the type information needed to safely use a module in a context, and consists of a sequence of *class type assignments*. A class type assignment is the type information on a single class declared in the module, and consists in a sequence of *constraints* and a *class signature*.

Constraints γ [1] express expectations on other classes used inside the class, and are formally defined and explained in the Appendix. Constraints may contain *type variables* α which denote arbitrary class names. For instance, $\phi(\mathsf{c}, \mathsf{f}, \alpha)$ means that class c is expected to provide a field named f of an arbitrary type. If the class for which this constraint has been inferred is then combined with a class named c providing a field named f of type, say, d, then the constraint is satisfied and the type variable α is instantiated to d.

A class type environment is a sequence of class signatures. Class signatures are the type information which can be extracted from a class declaration and consist of a simple class name (the name of the declared class), a class name (the name of the parent class), a sequence of *field signatures* (type and name of declared fields) and a sequence of *method signatures* (return type, name and parameter types of declared methods). The formal definition is provided in Fig.8 in the Appendix.

Typing rules are given in Fig.5.

$$(mdecs)\frac{\mathcal{M} \vdash \mathtt{MD}_i : (\mathtt{M}_i, \mathtt{MT}_i) \forall\ i \in 1..n}{\vdash \mathtt{MDS} : \mathcal{M}} \quad \begin{array}{l} \mathtt{MDS} = \mathtt{MD}_1 \ldots \mathtt{MD}_n \\ \mathcal{M} = (\mathtt{M}_1, \mathtt{MT}_1) \ldots (\mathtt{M}_n, \mathtt{MT}_n) \\ \rightarrow^+_{\mathtt{MDS}}\ \text{acyclic} \end{array}$$

$$(mdec)\frac{\mathcal{M} \vdash \mathtt{ME} : \mathtt{MT}}{\mathcal{M} \vdash \mathtt{module\ M\ is\ ME} : (\mathtt{M}, \mathtt{MT})}$$

$$(mname)\frac{}{\mathcal{M} \vdash \mathtt{M} : \mathtt{MT}_i} \quad \begin{array}{l} \mathcal{M} = (\mathtt{M}_1, \mathtt{MT}_1) \ldots (\mathtt{M}_n, \mathtt{MT}_n) \\ \mathtt{M}_i = \mathtt{M} \end{array}$$

$$(basic)\frac{\begin{array}{c} \vdash \mathtt{cd}_i : \Gamma_i \mid \delta_i\ \forall i \in 1..n \\ \Delta \vdash \Gamma_i \rightsquigarrow \Gamma_i'\ \forall i \in 1..n \\ \Delta\ \delta_1 \ldots \delta_n \vdash \Gamma_1' \ldots \Gamma_n' \diamond \end{array}}{\mathcal{M} \vdash \mathtt{cd}_i^{i \in 1..n} : (\Gamma_i', \delta_i)^{i \in 1..n}} \quad \Delta = ctenv(\mathcal{M})$$

$$(merge)\frac{\begin{array}{c} \mathcal{M} \vdash \mathtt{ME}_1 : \mathtt{MT}_1 \\ \mathcal{M} \vdash \mathtt{ME}_2 : \mathtt{MT}_2 \\ \Delta\ \Delta_1\ \Delta_2 \vdash \Gamma \diamond \end{array}}{\mathcal{M} \vdash \mathtt{merge}(\mathtt{ME}_1, \mathtt{ME}_2) : \mathtt{MT}_1\ \mathtt{MT}_2} \quad \begin{array}{l} \Delta = ctenv(\mathcal{M}) \\ \Delta_1 = ctenv(\mathtt{MT}_1) \\ \Delta_2 = ctenv(\mathtt{MT}_2) \\ \Gamma = cnstrs(\mathtt{MT}_1\ \mathtt{MT}_2) \\ out(\mathtt{MT}_1) \cap out(\mathtt{MT}_2) = \emptyset \end{array}$$

$$(restrict)\frac{\mathcal{M} \vdash \mathtt{ME} : (\Gamma_1, \delta_1) \ldots (\Gamma_n, \delta_n)}{\mathcal{M} \vdash \mathtt{restrict}(\mathtt{ME}, \mathtt{c}) : (\Gamma_1, \delta_1) \ldots (\Gamma_{i-1}, \delta_{i-1})(\Gamma_{i+1}, \delta_{i+1})(\Gamma_n, \delta_n)} \quad out(\delta_i) = \mathtt{c}$$

$$(rename)\frac{\mathcal{M} \vdash \mathtt{ME} : \mathtt{MT}}{\mathcal{M} \vdash \mathtt{rename}(\mathtt{ME}, \mathtt{c} \rightarrow \mathtt{c}') : \mathtt{MT}[\mathtt{c}'/in\ \mathtt{c}][\mathtt{c}'/out\ \mathtt{c}]} \quad \begin{array}{l} \mathtt{c} \in in(\mathtt{cds}) \cup out(\mathtt{cds}) \\ \mathtt{c}' \notin in(\mathcal{M}) \cup out(\mathcal{M}) \end{array}$$

$$(bind)\frac{\begin{array}{c} \mathcal{M} \vdash \mathtt{ME} : \mathtt{MT} \\ \Delta\ \Delta' \vdash \Gamma \diamond \end{array}}{\mathcal{M} \vdash \mathtt{bind}(\mathtt{ME}, \mathtt{d} \rightarrow \mathtt{c}) : \mathtt{MT}[\mathtt{c}/in\ \mathtt{d}]} \quad \begin{array}{l} \Delta = ctenv(\mathcal{M}) \\ \Delta' = ctenv(\mathtt{MT}[\mathtt{c}/in\ \mathtt{d}]) \\ \Gamma = cnstrs(\mathtt{MT}[\mathtt{c}/in\ \mathtt{d}]) \\ \mathtt{d} \in in(\mathtt{MT}) \\ \mathtt{c} \in out(\mathtt{MT}) \end{array}$$

$$(unbind)\frac{\mathcal{M} \vdash \mathtt{ME} : \mathtt{MT}}{\mathcal{M} \vdash \mathtt{unbind}(\mathtt{ME}, \mathtt{c} \rightarrow \mathtt{d}) : \mathtt{MT}[\mathtt{d}/in\ \mathtt{c}]} \quad \begin{array}{l} \mathtt{c} \in out(\mathtt{MT}) \\ \mathtt{d} \notin in(\mathtt{MT}) \cup out(\mathtt{MT}) \end{array}$$

Fig. 5. Typing rules

In rule (mdecs), a sequence of module declarations is well-formed and produces a given module type environment if under this module type environment each module declaration is well-formed and has the assigned module type. Note that this metarule is recursive to allow mutual references to module components, as in the following example:

```
module M1 is class C { M2.D f; ...}
module M2 is class D { M1.C g; ... }
```

However, mutual references to module names, as in

```
module M1 is rename C by D in M2
module M2 is rename D by C in M1
```

obviously make no sense, hence must be rejected by the type system. To this end, the side condition in rule (mdecs) requires the relation $\rightarrow_{\mathtt{MDS}}^{+}$ to be acyclic, where $\mathtt{M_1} \rightarrow_{\mathtt{MDS}} \mathtt{M_2}$ if there is in $\mathtt{MDS}$ a declaration $\mathtt{module\ M_1\ is\ ME}$ with $\mathtt{M_2}$ subterm of $\mathtt{ME}$, and $\rightarrow_{\mathtt{MDS}}^{+}$ denotes the transitive closure of $\rightarrow_{\mathtt{MDS}}$.

Rule (mdec) and (mname) are straightforward. We denote by $mname(\mathtt{MT})$ the first component of a module type (the module name).

In rule (basic), the type of a basic module (sequence of class declarations) can be inferred by the following steps:

- Each class declaration is separately type-checked, obtaining the constraints on other classes and the class signature. The formal definition of the judgment $\vdash \mathtt{cd} : \Gamma \mid \delta$, introduced in [1], is reported in Fig.8 in the Appendix.
- For each declared class, it is checked that constraints on qualified class names, that is, class names of the form $\mathtt{M.c}$, are satisfied by components of other modules, using their type information provided by the module type environment $\mathcal{M}$, and these constraints are removed. Formally, constraints Γ_i are simplified to Γ_i' under the class type environment Δ which can be extracted from $\mathcal{M}$. Note that a class signature for $\mathtt{M.c}$ can be extracted from the module type of $\mathtt{M}$ in $\mathcal{M}$ only if $\mathtt{M}$ has a $\mathtt{c}$ component and, moreover, is closed. This is expressed by the formal definition of $ctenv$ in Fig.6, where $in(\mathtt{MT})$ and $out(\mathtt{MT})$ denotes the deferred and defined class names appearing in $\mathtt{MT}$, respectively (defined class names are those appearing as first components of class signatures), and $\delta[\mathtt{c'}/\mathtt{c}]$ denotes replacement of all occurrences of $\mathtt{c}$ by $\mathtt{c'}$ in δ. For the formal definition of the judgment $\Delta \vdash \Gamma \rightsquigarrow \Gamma'$ we refer to [1].
- Then, it is checked that mutual constraints are satisfied by the class declarations in the module. Formally, constraints $\Gamma_1' \ldots \Gamma_n'$ left from the previous step are checked under the class type environment consisting of Δ and the class signatures extracted from the class declarations. Note that these constraints are checked but not simplified, since, as already explained, they need to be checked again in case a class declaration will be later overriden. The notation $\Delta \vdash \Gamma\diamond$ is an abbreviation for $\Delta \vdash \Gamma \rightsquigarrow \Gamma'$ for some Γ'.

$ctenv((\mathtt{M}_1, \mathtt{MT}_1) \ldots (\mathtt{M}_n, \mathtt{MT}_n)) = ctenv((\mathtt{M}_1, \mathtt{MT}_1)) \ldots ctenv((\mathtt{M}_n, \mathtt{MT}_n))$
$ctenv((\mathtt{M}, \mathtt{MT})) = ctenv(\mathtt{MT})[\mathtt{M}.\mathtt{c}_i / \mathtt{c}_i^{i \in 1..n}]$ where $\{\mathtt{c}_1, \ldots, \mathtt{c}_n\} = out(\mathtt{MT})$ if $in(\mathtt{MT}) = \emptyset$,
$ctenv((\mathtt{M}, \mathtt{MT})) = \Lambda$ if $in(\mathtt{MT}) \neq \emptyset$
$ctenv((\Gamma_1, \delta_1) \ldots (\Gamma_n, \delta_n)) = \delta_1 \ldots \delta_n$

$cnstrs((\Gamma_1, \delta_1) \ldots (\Gamma_n, \delta_n)) = \Gamma_1 \ldots \Gamma_n$

Fig. 6. Functions *ctenv* and *cnstrs*

In rule (merge), the operator can be safely applied only if the arguments have no conflicting class declarations, and, moreover, mutual constraints are satisfied by the class declarations in the two modules. Formally, constraints required by either argument are checked under the class type environment consisting of Δ and the class signatures extracted from both module types. The resulting module type is simply obtained by putting together the module types of the arguments. As in (basic), constraints are checked but not simplified.

In rule (restrict), the operator can be safely applied only if the class to be removed actually is a defined class of the module. The resulting module type is obtained by removing the corresponding class type assignment from the module type of the argument.

In rule (rename), the operator can be safely applied only if the class to be renamed is either a deferred or defined class of the module, and the new name is unused. The resulting module type is obtained by correspondingly renaming the module type of the argument.

In rule (bind), the operator can be safely applied only if a deferred class name $\mathtt{d}$ is bound to a defined class name $\mathtt{c}$, and, moreover, constraints which previously involved $\mathtt{d}$ are satisfied by $\mathtt{c}$. Formally, constraints extracted from the module type, where $\mathtt{d}$ has been replaced by $\mathtt{c}$, are checked under the class type environment consisting of Δ and the class signatures extracted from the module type, where also $\mathtt{d}$ has been replaced by $\mathtt{c}$. The resulting module type is obtained by replacing $\mathtt{d}$ by $\mathtt{c}$ in the module type of the argument. As in (basic) and (merge), constraints are checked but not simplified.

In rule (unbind), the operator can be safely applied only if a defined class name $\mathtt{c}$ is unbound, by replacing all its input occurrences by a fresh name $\mathtt{d}$. The resulting module type is obtained by analogously replacing all input occurrences of $\mathtt{c}$ by $\mathtt{d}$ in the module type of the argument (in analogy to what we have defined for class declarations, in a module type output occurrences are only those appearing as first components of class signatures).

In summary, the only operators which require a check of constraints are basic module, merge and bind, since these are the cases in which class names which were deferred (or just taken in isolation in the case of basic module) can become defined, hence some new type information on them can be provided which could

23

possibly contradict some required constraints. Restrict corresponds to remove some constraints, rename and unbind just to a syntactic manipulation of them.

Also note that constraint checking in the above rules is sometimes redundant, due to the fact that, as already explained, constraints are checked but not simplified in rules (basic), (merge), and (bind). An alternative type system could also keep trace, for each module, of the simplified constraints (that is, those which cannot be checked inside the module itself, since they still involve deferred classes). This would allow a more efficient approach in these rules: for instance, in rule (merge) it would be enough to check the simplified constraints of both modules, whereas in the rule presented here also constraints expressing internal consistency of both modules are checked again; however, with this approach it would be necessary to calculate again simplified constraints in rules (restrict) and (unbind), whereas no constraint simplification step is required for these rules in the current approach.

The relevance of the type system presented until now is that it supports *compositional compilation* of modules. This means that it is possible for the programmer to write and compile a SMARTJAVAMOD module in isolation[11], and then to combine it with other modules by just checking that mutual assumptions are satisfied, without any need of reinspecting code. In this sense, SMARTJAVAMOD is as an expressive module language layer constructed on top of the type system in [1]. Here only compositional compilation of single classes and *fragments*, consisting of either a single class or, inductively, a concatenation of fragments (as a matter of fact, an implicit *merge* operator), was considered.

Of course, compositional compilation can be safely used only in place of *global* compilation (that is, compilation of a sequence of classes in a context where all used classes are available in either source or binary form, as standard Java compilers do) only if module typing rules guarantee that, whenever by module composition we obtain in the end a closed module expression, then this module expression reduces to a well-typed self-contained sequence of classes. This is formally expressed, in a more general formulation which also takes into account module environment, by Theorem 3 below, which, as usually, can be proved by means of subject reduction and progress properties stated in Theorem 1 and 2, respectively.

Theorem 1 (Subject reduction).

- *If* $\vdash MDS : \mathcal{M}$, $MDS \to MDS'$, *then* $\vdash MDS' : \mathcal{M}$.
- *If* $\vdash MDS : \mathcal{M}$, $\mathcal{M} \vdash ME : MT$, $ME \to_{MDS} ME'$, *then* $\mathcal{M} \vdash ME' : MT$.

We say that a module declaration `module M is ME` is a *basic module declaration* if `ME` is a basic module; we say that a module environment $MD_1 \ldots MD_n$ is a *basic module environment* if each MD_i is a basic module declaration. For `MDS` basic module environment, we denote by *cenv*(`MDS`) the sequence of class declarations consisting of all the class declarations in some closed module, say `M`, where simple

---

[11] That is, in a context where classes used inside the module are not available; however, note that used *modules* must be available.

class names have been qualified by M, as formally defined in Fig.7. Analogously, $mt(\mathcal{M})$ denotes the sequence of class type assignments extracted from a module type environment.

$cenv(\texttt{MD}_1 \ldots \texttt{MD}_s) = cenv(\texttt{MD}_1) \ldots cenv(\texttt{MD}_n)$
$cenv(\texttt{module M is cds}) = \texttt{cds}[\texttt{M.c}_i/\texttt{c}_i^{i \in 1..n}]$ where $\{\texttt{c}_1, \ldots, \texttt{c}_n\} = out(\texttt{cds})$ if $in(\texttt{cds}) = \emptyset$
$cenv(\texttt{module M is cds}) = \Lambda$ if $in(\texttt{cds}) \neq \emptyset$
$mt((\texttt{M}_1, \texttt{MT}_1) \ldots (\texttt{M}_n, \texttt{MT}_n)) = mt(\texttt{M}_1, \texttt{MT}_1) \ldots mt(\texttt{M}_n, \texttt{MT}_n)$
$mt(\texttt{M}, \texttt{MT}) = \texttt{MT}[\texttt{M.c}_i/\texttt{c}_i^{i \in 1..n}]$ where $\{\texttt{c}_1, \ldots, \texttt{c}_n\} = out(\texttt{MT})$ if $in(\texttt{MT}) = \emptyset$
$mt(\texttt{M}, \texttt{MT}) = \Lambda$ if $in(\texttt{MT}) \neq \emptyset$

Fig. 7. Functions $cenv$ and mt

Theorem 2 (Progress).

- If $\vdash MDS : \mathcal{M}$, MDS not a basic module environment, then $MDS \to MDS'$ for some MDS'.
- If $\mathcal{M} \vdash ME : MT$, ME not a basic module, then $ME \to_{MDS} ME'$ for some ME'.

Theorem 3. If $\vdash MDS : \mathcal{M}$, $\mathcal{M} \vdash ME : MT$, $in(MT) = \emptyset$, then $MDS \to^\star MDS'$ with MDS basic module environment, $ME \to^\star_{MDS} \texttt{cds}$, $\Lambda \vdash cenv(MDS') \texttt{cds} : mt(\mathcal{M}) MT$ with $in(mt(\mathcal{M}) MT) = \emptyset$.

Theorem 3 states that, starting from a well-formed closed module expression in a well-formed module environment, we can always obtain a well-formed closed sequence of class declarations, consisting, roughly speaking, of the class declarations we get by reducing the module expression, together with those obtained by "flattening" class declarations in closed modules in the environment. Hence, the reduction semantics of an FJ expression in the context of a closed module expression ME and a module environment MDS can be defined as the reduction semantics of the expression in the context of the FJ program obtained by reducing ME and reducing and flattening MDS, and the result above guarantees type safety.

6 Implementation issues

In this section we discuss how a prototype compiler for SmartJavaMod could be implemented.

One of the key points of our approach is the ability of compiling Java classes and interfaces[12] in total isolation, without any burden on the programmers' side. The idea, as we already said, is to generate polymorphic bytecode, along with

---

[12] Not to be confused with module interfaces.

the type constraints representing the inferred requirements. We refer to [1] for a more detailed discussion of this topic.

A basic module is a collection of one or more classes/interfaces, which can be defined in one single file, as the examples seen so far may have suggested, or in many files, using one module definition file which "points to" standard Java sources (this feature comes handy to include plain Java classes in SMARTJAVAMOD modules).

A non basic module is defined in a single file which contains the association between the module name and its defining module expression.

When compiled, all kinds of modules share the same file format. The binary form of compiled modules is very similar to Jiazzi's one [24]; that is, a compiled module is an archive (a JAR file) which contains:

- the binary form of all classes/interfaces contained in the module, and
- the type constraints needed to ensure a type-safe linking.

If a module is open, then binary classes consist of polymorphic bytecode which cannot be directly loaded, much less executed, by a standard JVM (Java Virtual Machine). This is not a problem though, as an open module is incomplete and could not be run anyway.

When modules are composed by module operators, the type constraints are checked for consistency and if no errors are detected, a new module is created. Unfortunately, as said in Section 3.5, some errors could go undetected as long as modules remains open. Luckily, verification of constraints is complete in case of closed modules [1]. Enhancing the verification algorithm in order to capture inconsistencies as soon as possible is a challenging task which deserves future work.

In a closed module binary classes are stored in plain standard bytecode, so a binary module can be loaded and executed on any JVM. Moreover, module names are mapped into Java package names so, for instance, a class C contained in a module called m can be referenced by any Java class using the name m.C. This makes possible to use a compiled SMARTJAVAMOD (closed) module with any Java compiler (because type constraints are simply ignored by standard tools).

A feature, which could be beneficial for a real-world implementation, is the ability to add, along inferred type constraints, some constraints explicitly specified by the programmer. While this feature is unnecessary from a strictly theoretical point of view, as the most general requirements are always inferred, it could come handy when programmers needs to enforce particular module interfaces in order to interact with legacy code. Fortunately, this feature is very easy to add, as the constraint checking algorithm does not care whether a constraint has been inferred or written by the programmer. So, the compiler just needs to add the specified constraints to inferred ones to implement this feature.

7   Conclusion

We have presented SMARTJAVAMOD, a language of mixin modules constructed on top of Java which supports compositional compilation [1]. The semantics is given by reduction into pure Java, and the type system guarantees that the collection of classes obtained in the end is a well-formed Java program.

We have obtained a module language which is extremely flexible and allows to express a variety of constructs supporting software reuse and extensibility.

There are, to our knowledge, few attempts at integrating a true module system with Java. They include the simple extension of the package system proposed in [7], Jiazzi [24], Keris [30] and our own work in [3]. Instead there are many proposals for adding expressiveness to Java-like languages by extending the language in various ways, e.g., by introducing mixin classes [2], and virtual classes [12]. On the other hand, there exist several languages supporting modules in the context of object-oriented programming with structured types [?,?]; however, nominal types, as in Java, introduce different problems.

Concerning type inference for Java/Javascript-like languages, we mention some recent work [6], which, however, considers a very different setting (object based with structural subtyping).

The approach presented here introduces the following key novelties. First, *compositional compilation* allows to typecheck a module in isolation, inferring the type constraints needed for safely using it in any context, without any need for the user to specify an interface. Secondly, the module language supports *mixin modules*, that is, module operators which allow to redefine components. Finally, this module layer allows to express a variety of constructs, including mixin classes and virtual classes, without changing the core level (that is, keeping pure Java as core language).

There are two main directions for further research we want to explore. On the one hand, we plan to investigate in a more formal way how derived constructs, such as mixin classes, virtual classes and traits, can be encoded in SMARTJAVAMOD.

On the other hand, much remains to be investigated on the side of the implementation, in particular if one wants (as it seems desirable) to obtain a running extension actually constructed on top of Java. In particular, as already said in the paper, the verification of constraints is complete only in case of closed modules, so enhancing the algorithm in order to capture inconsistencies as soon as possible deserves future work.

Furthermore, a drawback of pure constraint inference is that there is no way to discover trivial typing errors, such as invoking the wrong method name, before linking since any reference to a missing component is just interpreted as a constraint. To avoid this, constraint inference should be in practice complemented by additional instruments to improve the practicality of the approach. For instance, compilers could print partial information extracted from the inferred constraints (which are heavy in their complete form) that can be checked by programmers, who should also be allowed to explicitly add constraints to the inferred ones.

References

1. D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Polymorphic Bytecode: Compositional Compilation for Java-like Languages. In *ACM Symp. on Principles of Programming Languages 2005*, January 2005.
2. D. Ancona, G. Lagorio, and E. Zucca. Jam—designing a Java extension with mixins. *ACM Transactions on Programming Languages and Systems*, 25(5):641–712, September 2003.
3. D. Ancona and E. Zucca. True modules for Java-like languages. In J. Knudsen, editor, *ECOOP 2001 - European Conference on Object-Oriented Programming*, number 2072 in Lecture Notes in Computer Science, pages 354–380. Springer, 2001.
4. D. Ancona and E. Zucca. A calculus of module systems. *Journ. of Functional Programming*, 12(2):91–132, 2002.
5. D. Ancona and E. Zucca. Principal typings for Java-like languages. In *ACM Symp. on Principles of Programming Languages 2004*, pages 306–317. ACM Press, January 2004.
6. C. Anderson, P. Giannini, and S. Drossopoulou. Towards Type Inference for JavaScript. In *ECOOP 2005 - Object-Oriented Programming*, Lecture Notes in Computer Science. Springer, 2005. To appear.
7. L. Bauer, A. Appel, and E. Felten. Mechanisms for secure modular programming in Java. Technical Report CS-TR-603-99, Department of Computer Science, Princeton University, July 1999.
8. A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems and Structures*, Sept. 2005.
9. V. Bono, A. Patel, and V. Shmatikov. A core calculus of classes and mixins. In R. Guerraoui, editor, *ECOOP'99 - European Conference on Object-Oriented Programming*, number 1628 in Lecture Notes in Computer Science, pages 43–66. Springer, 1999.
10. G. Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.
11. G. Bracha and W. Cook. Mixin-based inheritance. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1990*, volume 25(10) of *SIGPLAN Notices*, pages 303–311. ACM Press, October 1990.
12. K. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *ECOOP'98 - European Conference on Object-Oriented Programming*, pages 523–549, 1998.
13. K. B. Bruce and J. N. Foster. LOOJ: Weaving LOOM into Java. *Lecture Notes in Computer Science*, 3086:389–413, 2004.
14. C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. Technical Report 04-01b, Iowa State University, Dept. of Computer Science, Dec. 2004. Accepted for publication, pending revision.
15. K. Crary, R. Harper, and S. Puri. What is a recursive module? In *PLDI'99 - ACM Conf. on Programming Language Design and Implementation*, 1999.
16. K. Fisher and J. Reppy. The design of a class mechanism for MOBY. In *PLDI'99 - ACM Conf. on Programming Language Design and Implementation*, pages 37–49, May 1999.
17. M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *PLDI'98 - ACM Conf. on Programming Language Design and Implementation*, pages 236–248, 1998.

18. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *ACM Symp. on Principles of Programming Languages 1998*, pages 171–183. ACM Press, 1998.
19. R. Harper and M. Lillibridge. A type theoretic approach to higher-order modules with sharing. In *ACM Symp. on Principles of Programming Languages 1994*, pages 127–137. ACM Press, 1994.
20. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, pages 132–146, November 1999.
21. G. Kiczales. AspectJ(tm): Aspect-oriented programming in Java. *Lecture Notes in Computer Science*, 2591:1, 2003.
22. X. Leroy. Manifest types, modules and separate compilation. In *ACM Symp. on Principles of Programming Languages 1994*, pages 109–122. ACM Press, 1994.
23. X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, May 2000.
24. S. McDirmid, M.Flatt, and W. Hsieh. Jiazzi: New age components for old fashioned Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2001)*. ACM Press, October 2001.
25. A. Mendhekar, C. Maeda, C. Lopes, C. V. Lopes, G. Kiczales, J. marc Loingtier, J. Irwin, and J. Lamping. Aspect-oriented programming, Sept. 1999.
26. B. Meyer. *Eiffel: The language*. Prentice Hall, 1992.
27. G. L. Steele and S. M. Laboratories. Compatible genericity with run-time types for the Java programming language, Aug. 2001.
28. C. Szyperski. Import is not inheritance. why we need both: Modules and classes. In O. L. Madsen, editor, *ECOOP'92 - European Conference on Object-Oriented Programming*, number 615 in Lecture Notes in Computer Science, pages 19–32. Springer, 1992.
29. M. Torgersen. The expression problem revisited. In M. Odersky, editor, *ECOOP 2004 - Object-Oriented Programming*, number 3086 in Lecture Notes in Computer Science, pages 123–143. Springer, 2004.
30. M. Zenger. Evolving software with extensible modules. In *International Workshop on Unanticipated Software Evolution*, Malaga, Spain, June 2002.

Typing rules for classes [1] are formally defined in Fig.8. Constraints have the following informal meaning:

- $t \le t'$ means "t is a subtype of t'".
- $\exists n$ means "n is defined".
- $\phi(t, f, t')$ means "t provides field f with type t'".
- $\mu(t, m, \bar{t}, (t', \bar{t}'))$ means "t provides method m applicable to arguments of type $\bar{t}$, with return type t' and parameters of type $\bar{t}'$.
- $\kappa(n, \bar{t}, \bar{t}')$ means "n provides constructor applicable to arguments of type $\bar{t}$, with parameters of type $\bar{t}'$".
- $n \sim t$ means "n and t are comparable" (this constraint is generated when compiling a cast).

Note that both the constraints $\mu(t, m, \bar{t}, (t', \bar{t}'))$ and $\kappa(n, \bar{t}, \bar{t}')$ implicitly include the constraint $\bar{t} \le \bar{t}'$.

We write *type*(fds) and *type*(mds) to denote the set of field signatures and the set of method signatures extracted from the field declarations fds and from the method declarations mds, respectively. The straightforward definition of *type* has been omitted.

$$\gamma ::= \mathsf{t} \le \mathsf{t}' \mid \exists \mathsf{n} \mid \phi(\mathsf{t},\mathsf{f},\mathsf{t}') \mid \mu(\mathsf{t},\mathsf{m},\bar{\mathsf{t}},(\mathsf{t}',\bar{\mathsf{t}}')) \mid \kappa(\mathsf{n},\bar{\mathsf{t}},\mathsf{t}') \mid \mathsf{n} \sim \mathsf{t} \quad \text{constraint}$$

$\mathsf{t} ::= \mathsf{n} \mid \alpha$	expression type
$\bar{\mathsf{t}} ::= \mathsf{t}_1 \ldots \mathsf{t}_n$	expression types
$\mathsf{fss} ::= \mathsf{fs}_1 \ldots \mathsf{fs}_n$	field signatures
$\mathsf{fs} ::= \mathsf{n}\ \mathsf{f}$	field signature
$\mathsf{mss} ::= \mathsf{ms}_1 \ldots \mathsf{ms}_n$	method signatures
$\mathsf{ms} ::= \mathsf{n}\ \mathsf{m}(\bar{\mathsf{n}})$	method signature
$\bar{\mathsf{n}} ::= \mathsf{n}_1 \ldots \mathsf{n}_n$	class names
$\bar{\alpha} ::= \alpha_1 \ldots \alpha_n$	type variables
$\Pi ::= (\mathsf{x}_1,\mathsf{n}_1) \ldots (\mathsf{x}_n,\mathsf{n}_n)$	local environment

$(class)\dfrac{\vdash \mathsf{fds} : \Gamma \qquad \mathsf{c} \vdash \mathsf{mds} : \Gamma'}{\vdash \texttt{class c extends n} \{\ \mathsf{fds}\ \mathsf{mds}\ \} : \Gamma, \Gamma', \exists\mathsf{n} \mid (\mathsf{c},\mathsf{n},\mathsf{fss},\mathsf{mss})} \quad \begin{array}{l} type(\mathsf{mds}) = \mathsf{mss} \\ type(\mathsf{fds}) = \mathsf{fss} \end{array}$

$(fields)\dfrac{\vdash \mathsf{fd}_i : \Gamma_i \qquad \forall i \in 1..n}{\vdash \mathsf{fd}_1\ \ldots \mathsf{fd}_n : \Gamma_1 \ldots \Gamma_n}\ n \ne 1 \qquad\qquad (field)\dfrac{}{\vdash \texttt{n f; } : \exists\mathsf{n}}$

$(methods)\dfrac{\mathsf{c} \vdash \mathsf{md}_i : \Gamma_i \qquad \forall i \in 1..n}{\mathsf{c} \vdash \mathsf{md}_1\ \ldots \mathsf{md}_n : \Gamma_1 \ldots \Gamma_n}\ n \ne 1$

$(method)\dfrac{\mathsf{x}_1{:}\mathsf{n}_1 \ldots \mathsf{x}_n{:}\mathsf{n}_n, \texttt{this:c} \vdash \mathsf{e} : \mathsf{t} \mid \Gamma}{\mathsf{c} \vdash \mathsf{n}_0\ \mathsf{m}(\mathsf{n}_1\ \mathsf{x}_1 \ldots \mathsf{n}_n\ \mathsf{x}_n)\ \{\texttt{return e;}\} : \Gamma, \mathsf{t} \le \mathsf{n}_0, \exists\mathsf{n}_i{}^{i \in 0..n}}$

$(parameter)\dfrac{\Pi = (\mathsf{x}_1,\mathsf{n}_1) \ldots (\mathsf{x}_n,\mathsf{n}_n) \quad \mathsf{x} = \mathsf{x}_i}{\Pi \vdash \mathsf{x} : \mathsf{n}_i \mid \Lambda} \qquad (field\ access)\dfrac{\Pi \vdash \mathsf{e} : \mathsf{t} \mid \Gamma}{\Pi \vdash \mathsf{e.f} : \alpha \mid \Gamma, \phi(\mathsf{t},\mathsf{f},\alpha)}\ \alpha\ \text{fresh}$

$(meth\ call)\dfrac{\begin{array}{c} \Pi \vdash \mathsf{e}_0 : \mathsf{t}_0 \mid \Gamma_0 \\ \Pi \vdash \mathsf{e}_i : \mathsf{t}_i \mid \Gamma_i \quad \forall i \in 1..n \end{array}}{\Pi \vdash \mathsf{e}_0.\mathsf{m}(\mathsf{e}_1 \ldots \mathsf{e}_n) : \beta \mid \Gamma_0 \Gamma_1 \ldots \Gamma_n, \mu(\mathsf{t}_0,\mathsf{m},\mathsf{t}_1 \ldots \mathsf{t}_n,(\beta,\bar{\alpha}))}\ \beta,\bar{\alpha}\ \text{fresh}$

$(new)\dfrac{\Pi \vdash \mathsf{e}_i : \mathsf{t}_i \mid \Gamma_i\ \forall i \in 1..n}{\Pi \vdash \texttt{new n}(\mathsf{e}_1,\ldots,\mathsf{e}_n) : \mathsf{n} \mid \Gamma_1 \ldots \Gamma_n, \kappa(\mathsf{n},\mathsf{t}_1 \ldots \mathsf{t}_n,\bar{\alpha})}\ \bar{\alpha}\ \text{fresh}$

$(cast)\dfrac{\Pi \vdash \mathsf{e} : \mathsf{t} \mid \Gamma}{\Pi \vdash (\mathsf{n})\mathsf{e} : \mathsf{n} \mid \Gamma, \mathsf{n} \sim \mathsf{t}}$

Fig. 8. Typing rules for classes