Journal of Universal Computer Science, vol. 11, no. 10 (2005), 1613-1644
submitted: 2/5/05, accepted: 23/9/05, appeared: 28/10/05 © *J.UCS*

Analyzing Module Diversity

Alexandre Bergel
(Software Composition Group, University of Bern, Switzerland
bergel@iam.unibe.ch/∼scg)

Stéphane Ducasse
(LISTIC — Language and Software Evolution Group, University of Savoie, France
stephane.ducasse@univ-savoie.fr)

Oscar Nierstrasz
(Software Composition Group, University of Bern, Switzerland
oscar@iam.unibe.ch/∼scg)

Abstract: Each object-oriented programming language proposes various grouping mechanisms to bundle interacting classes (*i.e.,* packages, modules, selector namespaces, etc). To understand this diversity and to compare the different approaches, a common foundation is needed. In this paper we present a simple module calculus consisting of a small set of operators over environments and modules. Using these operators, we are then able to specify a set of module combinators that capture the semantics of Java packages, C# namespaces, Ruby modules, selector namespaces, gbeta classes, classboxes, MZScheme units, and MixJuice modules. We develop a simple taxonomy of module systems, and show how particular combinations of module operators help us to draw sharp distinctions between classes of module systems that share similar characteristics.

Key Words: package, module, selector namespaces, classboxes, virtual classes, Smalltalk, Java, Ruby, C#
Category: D.1.5, D.2.1, D.2.2, D.3.2, D.3.3

1 Introduction

Object-oriented languages support the construction of applications based on sets of interacting classes. Classes, methods and global definitions are then grouped together as packages or modules for deployment reasons or for delimiting abstraction layers. Unfortunately, though the intent behind packages and modules is clear, their semantics often is not. The simple fact that the terms *module* and *package* are overloaded with different semantics reveals a larger problem: the diversity of grouping mechanisms hampers their comparison and understanding.

Numerous researchers have proposed module calculi as a medium for reasoning about the properties of module systems (*e.g.,* [2, 8, 11, 31, 32]). These calculi have been used to study various issues such as mutual recursion and high order features [2] or to model classes and mixins in a typed setting [8]. As far as we are aware, however, module calculi have not yet been proposed as a means to study and compare the diverse approaches to module systems that are used in practice.

For example, classical module systems, like those of Modula-3 [16], Oberon-2 [36], Ada [40], Java [28], C++, C# [14], and Eiffel [33] do not support class extensions (*i.e.,* the fact that a method can be added or redefined from another package). However, class extensions are widely used in the languages that support it, such as Smalltalk [44], CLOS [29] and gbeta [21]. OpenClasses [17], Keris [45] and MixJuice [25] offer packaging systems that introduce class extensions, virtual classes and other new features to packages.

Other languages such as Ruby and Unit [24] support the definition and application of mixins to modules at different levels. Languages such as VisualWorks [43] totally decouple the issues of namespaces from those of code packaging, hence a package in VisualWorks does not provide any support for scoping of names.

In this paper we introduce a simple calculus of modules, together with a set of operators designed to express various encapsulation policies, composition rules, and extensibility mechanisms. Our work does not include deployment mechanisms, therefore the study of each language is limited in expressing operators applicable to grouping units (*i.e.,* modules, packages). The contributions of this paper are:

– A *formalism* for expressing semantics of different module systems,

– The identification of a set of *properties* useful to characterize different languages, and

– A *taxonomy* of different module systems.

The goal of the approach presented in this paper is to enable the language designer to compare features of module systems for various object-oriented programming languages.

In Section 2 we define the calculus and its operators. In Sections 3 through 10 we use the calculus to develop various module combinators that capture Java packages, C# namespaces, Ruby modules, selector namespaces, gbeta virtual classes, classboxes, MZScheme units and MixJuice modules. We chose Java and C# as they are mainstream languages, Ruby as it defines the notion of module mixin, Modular Smalltalk [44] and Smallscript [39] as they define changes that crosscut classes with selector namespaces, Beta [30] as it introduces the notion of virtual classes, classboxes [6, 7] as they illustrate the *local rebinding* property, MZScheme unit [23] as it separates the unit definition from the dependency statements, and MixJuice [25] as it constrains only one class version to be present in the system.

In Section 11 we develop a taxonomy to characterize the studied module systems according to the set of properties we modeled. In Section 12 we describe related work. In Section 13 we conclude by summarizing the results obtained and outlining future work.

2 A Simple Module Calculus

The module calculus we propose makes use of the primitive notion of an *environment*. In Section 2.1 we define environments and the basic operators for manipulating them. In Section 2.2 we define modules as abstractions over environments, and we propose combinators for composing and manipulating modules.

2.1 Environments

Definition 1 (Environment). An *environment* $\epsilon : D \rightarrow R^\star$, is a mapping from some domain D to an extended range $R^\star = R \cup \{\perp\}$, such that the inverse image $\epsilon^{-1}(R)$ is finite. The set of environments is $\mathcal{E}$ and we assume it to be a subset of R.

We will represent environments as finite sets of mappings, for example:

$$\epsilon_1 = \{a \mapsto x, b \mapsto y\}$$

is an environment that maps a to x and b to y. All other values in the domain of this environment (for example, $c \notin \{a, b\}$) are mapped to $\perp$.

We will normally leave out unessential parentheses. Since an environment is a function, we simply invoke it to look up a binding. In this case, $\epsilon_1\ a = x$, $\epsilon_1\ b = y$ and $\epsilon_1\ c = \perp$.

Definition 2 (Keys). The set of *keys* of an environment $\epsilon : D \rightarrow R^\star$ is $\kappa\ \epsilon$:

$$\kappa\ \epsilon \stackrel{\text{def}}{=} \{x \in D \mid \epsilon\ x \neq \perp\}$$

For instance, $\kappa\ \epsilon_1 = \{a, b\}$.

Definition 3 (Override). An environment $\epsilon : D \rightarrow R^\star$ may *override* another environment ϵ'. We define $\epsilon \rhd \epsilon' : D \rightarrow R^\star$ as follows:

$$(\epsilon \rhd \epsilon')\ x \stackrel{\text{def}}{=} \begin{cases} \epsilon'\ x & \text{if } \epsilon\ x = \perp \\ \epsilon\ x & \text{otherwise} \end{cases}$$

Note that $\rhd$ is associative but not symmetric.

For example, if $\epsilon_2 = \{b \mapsto z, c \mapsto w\}$, then $(\epsilon_1 \rhd \epsilon_2)\ a = x$, $(\epsilon_1 \rhd \epsilon_2)\ b = y$, and $(\epsilon_1 \rhd \epsilon_2)\ c = w$.

Definition 4 (Extend). An environment $\epsilon : D \rightarrow R^\star$ such that $\epsilon\ a = \bot$ may be *extended* to produce a new environment $\epsilon \| \{a \mapsto x\}$ containing all the mappings of ϵ plus $a \mapsto x$.

$$\epsilon \| \{a \mapsto x\} \stackrel{\text{def}}{=} \begin{cases} \epsilon \rhd \{a \mapsto x\} & \text{if } \epsilon\ a = \bot \\ \bot & \text{otherwise} \end{cases}$$

Note that the $\|$ operator does not allow an entry to be added if the key is already present.

Definition 5 (Exclusion). Given an environment $\epsilon : D \rightarrow R^\star$ and a key a, the *exclusion* $\epsilon \backslash a$ is:

$$(\epsilon \backslash a)\ x \stackrel{\text{def}}{=} \begin{cases} \epsilon\ x & \text{if } x \neq a \\ \bot & \text{otherwise} \end{cases}$$

Exclusion simply removes any binding present for the given key.

2.2 Modules

Definition 6 (Module). A *module* $m : \mathcal{E} \rightarrow \mathcal{E}^\star$, is a mapping from an environment to an environment. We denote $\mathcal{E}^\star = \mathcal{E} \cup \{\bot\}$ and $\mathcal{M}$ the set of modules.

Example. We represent modules as functions taking an environment and returning an environment. An example of two modules m_1 and m_2 follows:

$$m_1 = \lambda\epsilon.\ \{a \mapsto 1, b \mapsto 2\}$$
$$m_2 = \lambda\epsilon.\ \{a \mapsto 3, b \mapsto \epsilon\ a\}$$

As we see in m_2, the ϵ parameter makes it possible for entries in a module to look up other bindings in the parameter environment. Shortly we will see how a module can be *instantiated* to an environment by taking its fixpoint with the *fix* operator. Therefore, an instantiated module can look up bindings in itself.

We overload the operators previously introduced, and from the context it is always clear whether we mean the environment or module operator.

Definition 7 (Override). A module m can *override* another module m' to produce $m\ \rhd\ m'$.

$$m\ \rhd\ m' \stackrel{\text{def}}{=} \lambda\epsilon.\ (m\ \epsilon)\ \rhd\ (m'\ \epsilon)$$

Note that m and m' both have access to the parameter ϵ, thus effectively merging the two modules. As is the case with the other operators, we unambiguously overload the operator $\rhd$ for modules, defining it in terms of $\rhd$ for environments. It is easy to check that module overriding, like environment overriding, is also associative but not symmetric.

Definition 8 (Extend). A module can be *extended* with a new mapping using the $\|$ operator:

$$m\|\{a \mapsto x\} \stackrel{\text{def}}{=} \begin{cases} \lambda\epsilon.\ (m\ \epsilon)\ \|\ \{a \mapsto x\} & \text{if } (m\ \epsilon)\ a = \bot \\ \bot & \text{otherwise} \end{cases}$$

Definition 9 (Exclusion). *Exclusion* on modules is expressed using $\backslash$:

$$m\backslash x \stackrel{\text{def}}{=} \lambda\epsilon.\ (m\ \epsilon)\backslash x$$

Restricting a key on a module removes the key from the environment that this module defines.

2.3 Module Encapsulation Operators

Module encapsulation is articulated by operators manipulating the set of keys visible from outside the module. We call this set the module's interface. Three operators are here involved: *fix* is used to instantiate a module, κ to extract all the keys (*i.e.,* defined names) of a module, and *hide* to remove one mapping from a module interface.

Definition 10 (Fix). A module $m : \mathcal{E} \rightarrow \mathcal{E}^\star$ can be *instantiated* to an environment as follows:

$$\mathit{fix}\, m \stackrel{\text{def}}{=} \mu\epsilon.m\epsilon$$

(We assume the usual definition of μ, where $\mu x.e$ reduces to $e[\mu x.e/x]$, so $\mathit{fix}\, m = m(\mathit{fix}\, m)$.)

Since m is a function from environments to environments, $\mathit{fix}\, m$ represents a fixpoint in which all the mappings provided by the module are made available to each other. This is analogous to Cardelli's use of fixpoints to model self-references in object-oriented languages [15].

Example. For instance, for the two modules $m_1 = \lambda\epsilon.\ \{a \mapsto 1, b \mapsto \epsilon\}$ and $m_2 = \lambda\epsilon.\ \{a \mapsto 2\}$ we have:

$$\begin{array}{l} (\mathit{fix}\ (m_2 \triangleright \quad\quad\quad m_1 \quad\quad\quad))\ b\ a = 2 \\ (\mathit{fix}\ (m_2 \triangleright \lambda\epsilon.\ \{b \mapsto (\mathit{fix}\, m_1)\ b\}\ ))\ b\ a = 1 \end{array}$$

In the first example, let $\varPhi = \mathit{fix}(m_2 \triangleright m_1)$. By unfolding the fixpoint, this gives $\varPhi = \{a \mapsto 2\} \triangleright \{a \mapsto 1, b \mapsto \varPhi\}$, so $\varPhi\ b\ a = \varPhi\ a = 2$. Since m_1 and m_2 are effectively merged, m_2's binding of a becomes visible within m_1.

In the second example, let $\varPhi_1 = (\mathit{fix}\, m_1) = \{a \mapsto 1, b \mapsto \varPhi_1\}$ and $\varPhi_2 = \mathit{fix}(m_2 \triangleright \lambda\epsilon.\ \{b \mapsto \varPhi_1\ b\}) = \{a \mapsto 2\} \triangleright \{b \mapsto \varPhi_1\}$. So $\varPhi_2\ b\ a = \varPhi_1\ a = 1$. Here m_1 is effectively closed, so merging it with m_2 has no effect on the binding of a.

Definition 11 (Keys). The set of *keys* of a module m is defined as:

$$\kappa\ m \stackrel{\text{def}}{=} \kappa\ (\mathit{fix}\ m)$$

Definition 12 (Hide). Given a module $m : \mathcal{E} \rightarrow \mathcal{E}^{\star}$, a binding to the key a can be removed from its interface using the *hide* operator:

$$\begin{aligned} \mathit{hide}\ a &\stackrel{\text{def}}{=} \lambda m.\ \lambda\epsilon.\ m\backslash a\ (\{a \mapsto (\mathit{fix}\ m)\ a\} \triangleright \epsilon) \\ \mathit{hide}\ \{x_1, x_2, \ldots, x_n\} &\stackrel{\text{def}}{=} (\mathit{hide}\ x_1)(\mathit{hide}\ \{x_2, \ldots, x_n\}) \end{aligned}$$

The hide operator makes a binding inaccessible from outside a module. It can still be accessed from within it (*cf.* example below). This is different from simply removing a binding, because there is no distinction between the interface and the implementation of a module.

Example. Hiding a binding of a module removes the entry from the module's interface, but the binding's value is still internally accessible through other bindings. For example, for the two modules $m = \lambda\epsilon.\ \{a \mapsto 1, b \mapsto \epsilon\}$, we have:

$$\begin{aligned} (\mathit{fix}\ (\mathit{hide}\ a)(m))\ a\ \ \ \ &= \bot \\ (\mathit{fix}\ (\mathit{hide}\ a)(m))\ b\ a &= 1 \end{aligned}$$

2.4 Class Definition

Within a module, a value associated to a key represents a *definition*. If the calculus is intended to express the semantics of a module system which is part of a procedural (or functional) language, values of bindings describe functions [1, 31]. In the rest of this paper we will focus only on expressing semantics of module systems for class-based object-oriented programming languages. Definitions contained within a module describe classes. A class is represented as an environment whose mappings define state and methods. We also do not model class instantiation. Note that we do not model the pseudo-variable *super* because this would add unnecessary complexity to the calculus and shift the focus of the paper. Moreover, *super* is not present in all the languages we consider here. We also do not model classes as modules because our goal is to study module operators, not class operators. An attempt to unify modules and classes would generate unnecessary confusion.

We define the following domains:

- The set of modules is represented by $\mathcal{M}$.
- The set of class names by $\mathcal{C}$. It represents keys of the mappings defining a module.
- The set of definitions defining the state and behavior of a class is denoted by $\mathcal{D}$.

Example. For instance a module containing $Point$ and $PointFactory$ classes can be defined as:

$$
\begin{array}{l}
\mathit{GraphicsModule} = \lambda\epsilon.\ \{ \\
\qquad \mathit{Point} \mapsto \{ \\
\qquad\qquad\qquad\qquad x \mapsto 0, \\
\qquad\qquad\qquad\qquad y \mapsto 0, \\
\qquad\qquad \mathit{moveBy} \mapsto \lambda dx.\ \lambda dy.\ \lambda \mathit{self}.\ \{ \\
\qquad\qquad\qquad\qquad\qquad x \mapsto \mathit{self}\ x + dx, \\
\qquad\qquad\qquad\qquad\qquad y \mapsto \mathit{self}\ y + dy\ \} \rhd \mathit{self}\}, \\
\qquad \mathit{PointFactory} \mapsto \{\ \mathit{newPoint} \mapsto \lambda \mathit{self}.\ \epsilon\ \mathit{Point}\ \}\ \}
\end{array}
$$

By accessing a binding within a given environment we obtain a copy of the bound value. New objects are therefore obtained by simply accessing a class in a fixed module. For instance, a new point is obtained by evaluating:

aPoint = (fix GraphicsModule) Point.

Messages are sent by passing a message name and arguments to an object. Note that a reference to self has to be provided. For instance, a point can be moved by performing: *aPoint moveBy 2 3 aPoint.*

The self-reference is supplied as a trailing argument. The reason for this is that we do not model a dynamic environment (*i.e.,* a runtime stack that would contain the self reference and other arguments).

Inner classes [26] treat their enclosing class as a kind of module that restricts the scope of their definition. However, inner classes are mainly used as a convenient mechanism for defining callback classes and other small helper classes. To the best of our knowledge, no major application has ever been developed using inner classes as the sole module mechanism. We therefore do not consider this form of modularization in our comparison and do not model classes that contain inner classes.

Within our calculus, inheritance over classes is expressed by the *extendClass* operator, defined below.

Definition 13 (ExtendClass). In a module m, a class c *extends* a superclass sup with a set of definitions d using the *extendClass* combinator defined as:

$$
\begin{array}{l}
\mathit{extendClass}\ :\ \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{C} \rightarrow \mathcal{D} \rightarrow \mathcal{M} \\
\mathit{extendClass} = \lambda m.\ \lambda \mathit{sup}.\ \lambda c.\ \lambda d.\ \lambda\epsilon.\ m\ \epsilon \| \{c \mapsto d \rhd (m\ \epsilon\ \mathit{sup})\}
\end{array}
$$

Class members are denoted by d. When creating a new class, these simply override definitions provided by the superclass ($m\ \epsilon\ sup$) by performing $d \rhd (m\ \epsilon\ sup)$.

Example. The previously described *GraphicsModule* is refined into a *ColoredGraphicsModule* as:

$$\mathit{ColoredGraphicsModule} = \mathit{extendClass}\ \mathit{GraphicsModule}\ \mathit{Point}\ \mathit{ColoredPoint}\ \mathit{ext}$$
$$\text{where } \mathit{ext} = \{\ \mathit{color} \mapsto \mathit{nil},\quad \mathit{setColor} \mapsto \lambda\ \mathit{newCol}.\ \lambda \mathit{self}.\ \{\mathit{color} \mapsto \mathit{newCol}\} \triangleright \mathit{self}\ \}$$

2.5 The Calculus in Action

The following sections illustrate the calculus by expressing the semantics of various module systems. For each of these module systems we use the calculus to define a set of module combinators that express the module composition mechanisms provided by the programming language in question. Depending on the language under consideration, a module may represent a Java package, a C# namespace, a selector namespace (Modular Smalltalk [44]), a Ruby module [42], a virtual pattern in gbeta [21], a classbox [6, 7], a MZScheme unit [23], or a MixJuice module [25].

A typical combinator is applied to a module m_t with some arguments m_s and some classnames c. We make use of the following conventions: (i) each combinator is expressed as a module generator (combining two modules together yielding a new module), (ii) a t subscript (*e.g.,* m_t) refers to a input "template module" to be modified: the result of applying the combinator is a modified copy of the template module, (iii) an s subscript (*e.g.,* m_s) refers to the module provided as argument.

3 Java

Java [4] classes are grouped within *packages*. Packages can be "composed" with each other by means of the *import* relationship. (In this paper, we do not consider the use of fully qualified names in Java or other languages, *i.e.,* a class name preceded by the name of the package.) We also did not model the CLASSPATH mechanism as it is not particularly relevant to the paper's focus on the expression of grouping unit operators. A package that imports a class from another package simply references this class by its name. There are two levels of granularity of the *import* relationship: (i) a package may import a single class from another package, and (ii) a package may import all visible classes (*i.e.,* public at the package level).

The semantics of Java packages is expressed using two different import combinators (*importClass* and *importPackage*) corresponding to the two granularity levels. Class privacy is expressed using the *private* combinator, which is described later.

Individual class import. A package m_t that imports a class c defined in a package m_s yields a copy of m_t augmented with a new mapping for the imported class.

$$importClass : \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{M}$$
$$importClass = \lambda m_t.\ \lambda m_s.\ \lambda c.\ (hide\ c)(m_t \parallel \{c \mapsto (fix\ m_s)\ c\})$$

A package that imports a class cannot re-export it. This is expressed by ($hide\ c$). According to Definition 12, ($hide\ c$) is a function that takes a module as argument and returns a copy of it where c is removed from its interface only. This class c is accessible to definitions in the importing package, however it cannot be accessed from another package.

A conflict occurs when a defined and an imported class have the same name. The restriction on key uniqueness is expressed by the $\parallel$ operator, which does not allow a key to be added if it is already present. A name that already refers to a defined class cannot be used to refer to an imported one and vice-versa.

References between classes are static, which means that importing a class does not rebind the references. Let's suppose a package *graphics* contains a class *PointFactory* that refers to a class *Point*. Importing the class *PointFactory* (in another package) does not impact the original references between *PointFactory* and *Point*, even if in the importing package a class *Point* is present. This restriction is expressed by ($fix\ m_s$). An example of a graphics package is:

```
package graphics;
public class Point {
    int x = 0, y = 0;
    void moveBy (int dx, int dy) {
        x = x + dx; y = y + dy;
    }
}
public class PointFactory {
    static void newPoint () {
        return new Point();
    }
}
```

$$\begin{array}{l} graphics = \lambda\ \epsilon.\ \{ \\ \quad Point \mapsto \\ \qquad \{x \mapsto 0, \\ \qquad y \mapsto 0, \\ \qquad moveBy \mapsto \lambda\ self.\ \lambda\ dx.\ \lambda\ dy. \\ \qquad\quad \{\ x \mapsto self\ x + dx, \\ \qquad\quad y \mapsto self\ y + dy\} \rhd self\}, \\ \quad PointFactory = \\ \qquad \{\ newPoint \mapsto \epsilon\ Point\ \}\} \end{array}$$

Importing the class PointFactory from graphics in a new package graphics2 where a class Point already exists does not make PointFactory use Point of graphics2.

$$graphics2 = importClass\ \lambda\epsilon.\ \{Point \mapsto \{\}\}\ graphics\ PointFactory$$

Evaluating *graphics2 PointFactory newPoint* returns an environment containing the keys x, y, and *moveBy*. However, evaluating *graphics2 Point* yields an environment with no mapping in it (according to the definition of *Point* in *graphics*2.

Individual package import. Importing a whole package is equivalent to importing individually each class defined in the imported module. In a Java program, this is expressed as `package mt; import ms.*;`.

$$importPackage \, : \, \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{M}$$
$$importPackage = \lambda m_t.\ \lambda m_s.\ (hide\ (\kappa\ m_s))(\lambda \epsilon.\ (m_t\ \epsilon) \rhd\ (fix\ m_s))$$

When importing a whole package, locally-defined classes mask the classes that are imported. For instance, the following code is correct:

```
package p1;
public class A{}
public class B{}
```

```
package p2;
import p1.*;
public class A extends B{}
```

In the package p2, a class named A is locally defined, which masks the class A implicitly imported from p1. The name A within p2 refers to the p2 implementation of A, whereas in p1 the name A refers to the p1 implementation of A.

This is expressed by the $\rhd$ operator (which allows one mapping to be replaced by a new one) used in *importPackage*.

Class privacy. Classes declared as private in a package cannot be imported to other packages.

$$private \, : \, \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{M}$$
$$private = \lambda m.\ \lambda c.\ (hide\ c)\ m$$

Syntactically this is written `package m; class C {...}`. The class `C` is visible only within `m` and is not accessible from outside.

4 C#

In C# a unit of modularization is called a *namespace*. Classes defined in a namespace can be imported under a different name (alias) in the importing namespace. C# provides a unique directive `using` to import a class (aliased or not) to a namespace and to import a whole namespace.

We express the semantics of the `using` directive with the three combinators *usingClassAs*, *usingClass* and *usingNamespace*.

Using alias and class directives. An imported class can be aliased. This means that this class is accessed within the importing namespace under a different name. This is expressed in C# as `namespace MT { using A = MS.C; }`.

$$usingClassAs \, : \, \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{C} \rightarrow \mathcal{M}$$
$$usingClassAs = \lambda m_t.\ \lambda m_s.\ \lambda a.\ \lambda c.\ (hide\ a)(m_t \| \{a \mapsto (fix\ m_s)c\})$$

The value a refers to the new name given to the class c. The need for ($hide\ a$) and ($fix\ m_s$) is similar to that in Java's case: ($hide\ a$) constrains the imported class to be imported from another module, and ($fix\ m_s$) binds all classes contained in m_s to their dependencies. Also, when importing c in m_t, dependencies of c are preserved.

Importing a class without renaming it is expressed as an import aliased to the class name. This is expressed in C# as `namespace MT { using MS.C;}`.

$$\begin{aligned} &usingClass\ :\ \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{M} \\ &usingClass = \lambda m_t.\ \lambda m_s.\ \lambda c.\ usingClassAs\ m_t\ m_s\ c\ c \end{aligned}$$

Note that *usingClass* is equivalent to *importClass* previously described for Java.

Using the namespace directive. As in Java, all the classes defined in a namespace can be imported using a single directive. In a C# program, this would be expressed as `namespace MT { using MS; }`.

$$usingNamespace = importPackage$$

The combinator *usingNamespace* is equivalent to *importPackage* described previously.

C# provides the extern alias keywords to enable references to two different versions of deployment units (*i.e.,* assemblies) that have the same fully-qualified type names. This allows two or more versions of a given deployment unit to be used in the same application. Since the focus of this paper is on code packaging mechanisms rather than application deployment, we do not consider this mechanism in our comparison.

5 Ruby

In Ruby, modules serve two distinct purposes: (i) modules encapsulate functions, methods, classes, and constants, and (ii) a module is a namespace of methods that can also be used as a mixin. In this section we therefore separately consider two operators, *includeModule* and *newClassWithMixin*, that respectively express these two purposes.

Modules as namespaces. A module m_t uses the code provided by another module m_s by means of an include directive. This directive takes as parameter the name of the module intended to be reused. In the following example a module named MPoint defines a class Point containing a constructor. Another module MColoredPoint imports the definitions of MPoint and defines a subclass

ColoredPoint of Point.

```
#Defined in a file MPoint.rb
module MPoint
    class Point
        def initialize(x, y)
            @x = x
            @y = y
            end end end
```

```
#Defined in a file MColorPoint.rb
load "MPoint.rb"
module MColoredPoint
    include MPoint
    class ColoredPoint < Point
        # ...
        end end
```

We define an *includeModule* combinator that expresses the semantics of this include relationship between two modules. The resulting module of this combinator is a merge between the two where (i) local definitions hide those of the imported module, and (ii) references of classes contained in the provider module are preserved.

$$\begin{aligned} &\mathit{includeModule} \ : \ \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{M} \\ &\mathit{includeModule} = \lambda m_t.\ \lambda m_s.\ \lambda \epsilon.\ m_t\ \epsilon \rhd (\mathit{fix}\ m_s) \end{aligned}$$

Definitions contained in the importing module m_t have precedence over those of the imported module m_s in case of duplicate definitions. Before including a module, this one needs to be fixed because references between classes in this module are preserved.

Modules as mixins. As defined by Bracha and Cook [10], a mixin is a subclass definition that may be applied to different superclasses to create a related family of modified classes. In Ruby, a *module mixin* is a set of methods intended to be used as part of the definition of a class. A module that defines only a set of functions can be used as a mixin within a class definition by means of an include statement. In that case all the functions defined in the module are methods applicable to any instance of the class.

The following example shows the definition of a module mixin named MColor and a class ColoredPoint that uses it:

```
#Defined in a file MColor.rb
module MColor
    def getColor()
        @color
    end
    def setColor(col)
        @color = col
    end
    def setToBlack()
        self.setColor("Black")
    end
end
```

```
load "MColor.rb"
class ColoredPoint
    include MColor
    def initialize (x, y)
        @x = x
        @y = y
        self.setToBlack()
    end
    def getX() @x end
    def getY() @y end
end
```

The two methods getColor() and setColor() can be invoked on instances created by the class ColoredPoint. For example, the following code yields 5 and Black.

```
@p = ColoredPoint.new(2,3)
puts @p.getX() + @p.getY(); puts @p.getColor()
```

The semantics of the include construct which treats a module as a mixin can be expressed within the calculus as follows:

$$\begin{aligned} newClassWithMixin &: \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{D} \rightarrow \mathcal{M} \\ newClassWithMixin &= \lambda m_t.\ \lambda mixin.\ \lambda c.\ \lambda d. \\ &\quad m_t \| \{c \mapsto \mathit{fix}(\lambda\sigma.\ d \triangleright mixin\ \sigma)\} \end{aligned}$$

The module mixin intended to be included in the class c is named *mixin*. The self-reference contained in the module *mixin* is rebound to the class being created by the *fix* operator.

A mixin can only be used by creating a new class. To represent this, we defined the *newClassWithMixin* combinator expressing the semantics of creating a new class composed of one mixin. To keep the model concise, we do not handle situations in which (i) a subclass includes some mixin, or (ii) a class can be composed of several mixins. These can easily be expressed by a combinator that would accept a superclass and a set of mixins.

6 Selector Namespaces

It is a tradition for Smalltalk and Lisp-based programming languages to offer a mechanism for introducing *class extensions* [6]. A class extension is a method addition or a redefinition applied to a class already present in a system. The result is an evolution of the behavior defined by this class without introducing

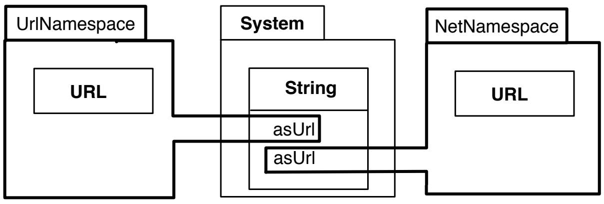

Figure 1: Two class extensions occur on the class String: two methods asUrl are added by two different namespaces UrlNamespace and NetNamespace.

a subclass. The intent of this mechanism is to enable better distribution of responsibility among the involved classes.

The concept of selector namespaces was first introduced in Modular Smalltalk [44], and more recently in Smallscript [39], a Smalltalk implementation for .Net. A selector namespace defines a namespace for methods and is used to manage conflicting class extensions. Within such a namespace one can extend any class in the system without producing conflict: another namespace can contain a class extension having the same name. This is illustrated by Figure 1 where the class String is extended by two namespaces UrlNamespace and NetPackage, each of them adding a method asUrl.

For instance, Figure 2 shows a class Object defined in a namespace English. This class contains a method printOnStream: and two printString methods. A first implementation of printString is provided by the selector namespace English, and the second one by German. A possible implementation is presented on the right side of Figure 2, where we see that the class Object contains three entries in its method dictionary. Each method has its name preceded by the name of the selector namespace that implements it. The method lookup is performed according to which namespace messages are sent from.

Sending the message printOnStream: within the selector namespace German results in the following steps: (i) look up for German.printOnStream:, (ii) German.printOnStream: does not exist, therefore, (iii) because German imports Object from English, lookup continues by searching for English.printOnStream:. (iv) This method is found and invoked.

Selector namespaces are non reentrant. If within a selector namespace a particular method is not found, then the lookup is pursued in the selector namespace from which the class is imported. However, a method implementation is always looked up according to the namespace within which the message is *actually sent*. For instance, invoking printOnStream: within the namespace German

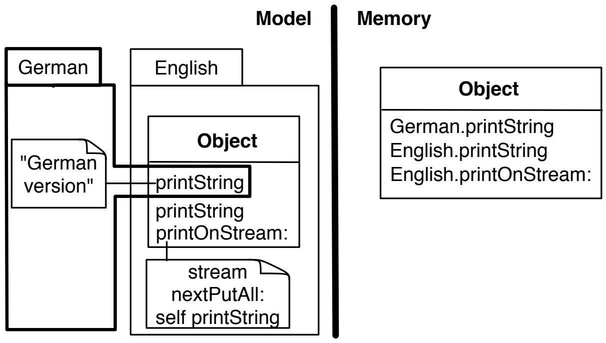

Figure 2: The class Object is composed of three methods: two versions of printString and a printOnStream: method.

triggers the method English.printOnStream:. This method triggers printString, also the implementation used is English.printString because the call for it occurs in the namespace English. Even if called from within German, the method English.printOnStream: *cannot* invoke German.printString. We qualify this lookup as *non reentrant*.

Importing and extending. Two combinators can be applied to a selector namespace: (i) import and extend a class (*extend*), or simply (ii) import (*import*) a class. Importing from a namespace m_s and extending a class c with a set of methods d is expressed with the *extend* combinator:

$$\begin{aligned} &\mathit{extend} : \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{M} \rightarrow \mathcal{M} \\ &\mathit{extend} = \lambda m_t.\ \lambda m_s.\ \lambda c.\ \lambda d.\ m_t \| \{c \mapsto (\mathit{fix}\ d) \rhd ((\mathit{fix}\ m_s)\ c)\} \end{aligned}$$

When a selector namespace extends a class, the extending methods need to keep a reference to the scope that contains them. In order to keep this reference, a set of methods is a module (note that for the above formula $d \in \mathcal{M}$) and it is fixed when used to extend a class *(fix d)*.

For instance, a namespace English containing a class Object (Figure 2) is defined as:

$$\begin{aligned} \mathit{English} = \mathit{extend}\ \lambda\epsilon.\ \{\}\quad &\mathit{Object} \\ &\lambda s.\ \{\mathit{printString} \mapsto \lambda \mathit{self}.\ \mathit{englishVersion}\}, \\ &\quad \mathit{printOnStream} \mapsto \lambda \mathit{self}.\ (s \rhd \mathit{self})\ \mathit{printString}\} \end{aligned}$$

This class Object is extended with a German implementation of the printString method. The German namespace is defined as:

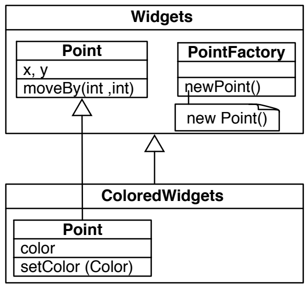

Figure 3: The outer class ColoredWidgets refines the class Point. Because classes are looked up, points produced from a factory obtained from ColoredWidgets are colored.

$$\begin{aligned} German = {} & extend\ \lambda\epsilon.\ \{\}\ English\ Object \\ & \lambda s.\ \{printString \mapsto \lambda self.\ germanVersion\} \end{aligned}$$

The second combinator associated to selector namespace is the *import*. A namespace imports a class without extending with:

$$\begin{aligned} &import\ :\ \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{M} \\ &import = \lambda m_t.\ \lambda m_s.\ \lambda c.\ extend\ m_t\ m_s\ c\ \ \lambda\epsilon.\ \{\} \end{aligned}$$

7 Virtual Classes

The notion of *virtual classes* offered by gbeta [22], Caesar [34] or Keris [45] allows class names to be dynamically looked up (rather than statically, at compilation time). Virtual classes unify the method and class lookup under a common lookup algorithm: as well as methods, class definitions are looked up along the inheritance of outer classes.

In gbeta, virtual classes are implemented as inner classes, and outer classes define the unit of modularization. Class names are looked up in the same way methods are looked up: inner classes can be refined within subclasses of the outer class.

Figure 3 shows the case where a set of inner classes contained in an outer class Widgets is refined in ColoredWidgets. The class Point defined in Widgets is subclassed into a new class Point in ColoredWidgets. The class PointFactory is visible into this last class because inherited. When the method newPoint() is triggered, the class Point is looked up according to the hierarchy of outer classes. If the factory is obtained from an instance of the outer class ColoredWidgets, then the points produced are colored.

Two operations modeling inheritance are involved when handling virtual classes: (i) inheritance between outer classes and (ii) inheritance between inner classes. Within our calculus the semantics of these two operations are expressed with the combinators *extendEncapsulated* and *extendInner*.

Elements in the subclass hide ones defined in the superclass. Inheritance between outer classes is simply expressed using the operator $\triangleright$. The *extendEncapsulated* combinator is defined as:

$$\begin{aligned} &\mathit{extendEncapsulated} : \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{M} \\ &\mathit{extendEncapsulated} = \lambda m_t.\ \lambda m_s.\ m_t \ \triangleright m_s \end{aligned}$$

Inheritance between inner classes is defined in a similar way that classical inheritance (Definition 13). The *extendInner* combinator is:

$$\begin{aligned} &\mathit{extendInner} : \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{D} \rightarrow \mathcal{M} \\ &\mathit{extendInner} = \lambda m_t.\ \lambda c.\ \lambda d.\ \lambda \epsilon.\ (m_t \backslash c)\ \epsilon \| \{c \mapsto d \triangleright\ (m_t\ \epsilon\ c)\} \end{aligned}$$

For instance, assuming an outer class *Widgets*, *ColoredWidgets* is defined as:

$$\begin{aligned} &\mathit{ColoredWidgets} = \mathit{extendInner}\ (\mathit{extendEncapsulated}\ \lambda\epsilon.\ \{\}\ \mathit{Widgets}) \\ &\qquad\qquad\qquad\qquad \mathit{Point}\ \mathit{colorExtensions} \\ &\text{where } \mathit{colorExtensions} = \{\mathit{color} \mapsto \mathit{black}, \\ &\qquad \mathit{setColor} \mapsto \lambda \mathit{newCol}.\ \lambda \mathit{self}.\{\mathit{color} \mapsto \mathit{newCol}\} \triangleright \mathit{self}\} \end{aligned}$$

8 Classboxes

The classbox model [6] is a module system that supports *local rebinding*. A class defined in one particular classbox can be extended via method addition or *redefinition* in other classboxes. The changes introduced by a classbox are *only* visible to the extending classbox and classboxes that import it. Moreover, redefined methods take precedence over existing ones *when invoked from an extending classbox*, and this even if the methods are called via methods only defined in the extended classbox. Whereas virtual classes (*i.e.,* as in gbeta) unify the lookup of methods and the lookup of classes under a common algorithm, classboxes offer a scoping mechanism to control the visibility of class extensions.

The following example illustrates a method extension with local rebinding [6]. Figure 4 depicts a classbox WidgetsClassbox that defines a class Morph, which is the root of the graphic element hierarchy in Squeak [27] (a smalltalk dialect in which classboxes are implemented), and a subclass Button. Morph contains a paint() method and a repaint() that calls paint(). The classbox EnhWidgetsClassbox imports Morph and redefines the paint() method. It also imports the subclass Button. In the context of WidgetsClassbox, invoking the repaint() method on an instance of Button invokes the definition of paint() in Morph defined by

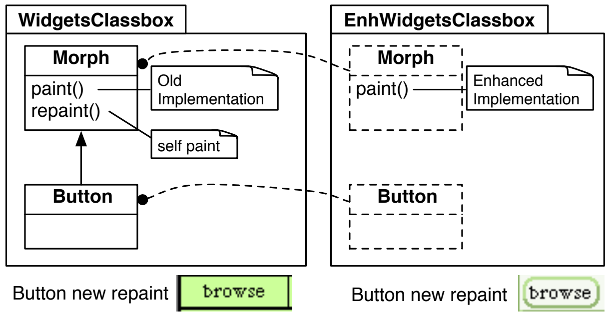

Figure 4: Implicitly rebinding classes within classboxes.

this classbox. Within EnhWidgetsClassbox, invoking repaint() triggers the new implementation of paint() defined in this classbox.

Within our calculus the semantics of classboxes are expressed using the *extend* operation. Within a classbox, a class may be imported from another classbox, and extended by new definitions.

This combinator is defined as:

$$extend\ :\ \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{D} \rightarrow \mathcal{M}$$
$$extend = \lambda m_t.\ \lambda m_s.\ \lambda c.\ \lambda d.\ \lambda \epsilon.\ m_t\ \epsilon \| \{c^{m_s} \mapsto d \rhd\ (m_s\ \epsilon\ \ c)\}$$

For instance, the extension between EnhWidgetsClassbox and WidgetsClassbox is stated:

$$EnhWidgetsClassbox\ =\ extend\ \lambda\epsilon.\ \{\}\ \ WidgetsClassbox\ Morph\ \{paint \mapsto\ \ldots\}$$

The superscript (*e.g.*, c^{m_s}) is used to identify the *originating classbox* in which a class is first defined. This makes it possible to distinguish classes that are defined from those that are imported and extended, even if they have the same name. Let's suppose that classbox WidgetsClassbox defines two classes Morph and Button, where Button makes use of Morph. If a classbox EnhWidgetsClassbox imports Button from WidgetsClassbox and defines a new class Morph, then this new class has nothing to do with the Morph originating in WidgetsClassbox and should not affect the imported Button class. The superscript identifying the originating classbox ensures that no confusion will result. If, on the other hand, EnhWidgetsClassbox imports and extends Morph from WidgetsClassbox, then this extended Morph will have the same originating classbox superscript, and will affect the imported Button class.

In the same way, new classes are defined as

$$newClass\ :\ \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{C} \rightarrow \mathcal{D} \rightarrow \mathcal{M}$$
$$newClass = \lambda m_t.\ \lambda sup.\ \lambda c.\ \lambda d.\ \lambda \epsilon.\ m_s\ \epsilon \| \{c^{m_t} \mapsto d \rhd \epsilon\ sup\}$$

A shortcut to extend a class with an empty set of definition is stated as:

$$\begin{aligned}&\mathit{import} : \mathcal{M} \to \mathcal{M} \to \mathcal{C} \to \mathcal{M}\\&\mathit{import} = \lambda m_t.\ \lambda m_s.\ \lambda c.\ \mathit{extend}\ m_t\ m_s\ c\ \{\}\end{aligned}$$

9 Units

MZScheme [23] offers an advanced module system based on units. A program *unit* is an unevaluated fragment of code intended to be linked with other units in order to form executable programs. There is no global namespace of units.

A unit describes its import requirements without specifying a particular unit that supplies those imports. The actual linking of the unit is specified externally at a later stage. Unlike in ML, unit linking is specified for groups of units with a graph of connections, which allows mutual recursion across unit boundaries. Furthermore, the result of linking a collection of units is a new (compound) unit that is available for further linking. One important point of this module system is that connections between modules are specified separately from their definitions.

The *link* combinator is defined as:

$$\begin{aligned}&\mathit{link} : \mathcal{M} \to \mathcal{M} \to \mathcal{C} \to \mathcal{C} \to \mathcal{M}\\&\mathit{link} = \lambda m_t.\ \lambda m_s.\ \lambda a.\ \lambda c.\ \lambda \epsilon.\ m_t\ \epsilon \| \{a \mapsto m_s\ \epsilon\ c\}\end{aligned}$$

Applying a *link* combinator to units m_t and m_s makes the value associated to c in m_s available in m_t under the alias a. For instance, Figure 5 shows a *widgets* unit defining two classes *Point* and *Circle* and a *widgetsFactory* unit defining a class *Factory*. The corresponding expression is:

$$\begin{aligned}\mathit{widgets} &= \lambda \epsilon.\ \{\mathit{Point} \mapsto \{\ x \mapsto \mathit{0},\ y \mapsto \mathit{0}\},\\&\qquad\quad \mathit{Circle} \mapsto \{\mathit{radius} \mapsto \mathit{0},\ \mathit{center} \mapsto \epsilon\ \mathit{Point}\}\}\\\mathit{widgetsFactory} &= \lambda \epsilon.\ \{\mathit{Factory} \mapsto \{\mathit{newPoint} \mapsto \epsilon\ \mathit{Point}\},\\&\qquad\qquad\qquad \mathit{newCircle} \mapsto \epsilon\ \mathit{Circle}\}\}\end{aligned}$$

To make *widgetsFactory* use the widgets a new compound *compound1* is created using:

$$\begin{aligned}\mathit{compound1} = \mathit{link}\ (&\mathit{link}\ \mathit{widgetsFactory}\ \mathit{widgets}\ \mathit{Point}\ \mathit{Point})\\&\mathit{widgets}\ \mathit{Circle}\ \mathit{Circle}\end{aligned}$$

The unit *compound1* is the result of linking classes *Point* and *Circle* defined in *widgets* under their original names (*Point* and *Circle*) in the unit *widgetsFactory.* This compound is obtained by linking the class *Point* obtained from *widgets* to the name *Point* in the unit *widgetsFactory.* Then, the class *Circle* of *widgets* is linked to the name *Circle* in *widgetsFactory.*

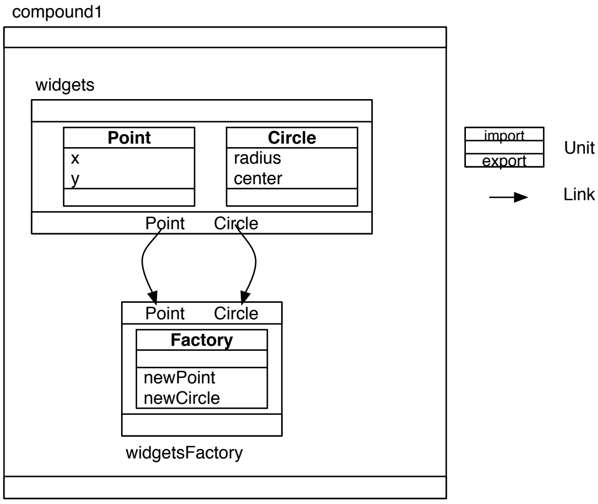

Figure 5: Composing two units, widgets and widgetsFactory, into one compound, compound1.

As illustrated in Figure 6, the widget factory is used by colored widgets, without altering the original definition of widgetsFactory. This is expressed with:

$$\begin{aligned}
\mathit{coloredWidgets} \;=\; \lambda\epsilon.\, \{ & \mathit{ColoredPoint} \mapsto \{\, \mathit{color} \mapsto \mathit{blue},\ x \mapsto \mathit{0},\ y \mapsto \mathit{0}\},\\
& \mathit{ColoredCircle} \mapsto \{\mathit{color} \mapsto \mathit{blue},\ \mathit{radius} \mapsto \mathit{0},\\
& \qquad \mathit{center} \mapsto \epsilon\ \mathit{Point}\}\}
\end{aligned}$$

$$\begin{aligned}
\mathit{compound2} = \mathit{link}\ (&\mathit{link}\ \mathit{widgetsFactory}\ \mathit{coloredWidgets}\ \mathit{ColoredPoint}\ \mathit{Point})\\
&\mathit{coloredWidgets}\ \mathit{ColoredCircle}\ \mathit{Circle}
\end{aligned}$$

The unit *compound2* is the result of linking the class *ColoredPoint* and *ColoredCircle* obtained from *coloredWidgets* to the unit *widgetsFactory* under the names *Point* and *Circle*.

10 MixJuice

MixJuice [25] is a module system for Java in which a module encapsulates the differences between the original program and the extended program. The difference is a set of definitions of additions and modifications of classes, fields and methods. Modules may inherit other modules. As explained below, an important distinction between the MixJuice module system and classboxes is that an original and a modified version of a set of classes *cannot* be present at the same time in the same system with MixJuice.

For instance, a module defining a point is defined as:

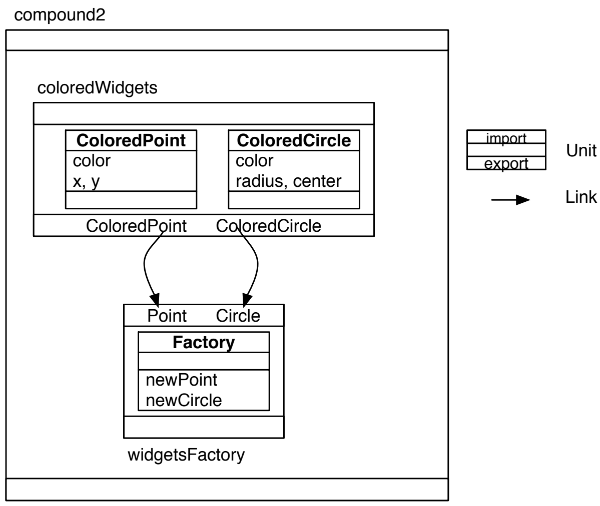

Figure 6: The unit widgetsFactory is composed with a new unit coloredWidgets.

```
module point {
    define class Point {
        define int x = 0;
        define int y = 0;
        define void moveBy (int dx, int dy) { x += dx; y+= dy;}
        define String toString () { return "point "+x+","+y;}
    }
}
```

The keyword define is used to define a new class member. Without this keyword, the class is refined. Here the class Point is refined to a colored point:

```
module coloredPoint extends point {
    class Point {
        define Color c; // Variable addition
        // Redefinition of the method toString()
        String toString () { return "colored point "+x+","+y;}
    }
}
```

The example above uses a single inheritance link between modules. However, multiple inheritance is permitted. In that case, all modules are linearized by topological sort (similar to the class linearization done in CLOS [19]).

Within our calculus, semantics of MixJuice modules are expressed using two combinators: *extends* to express inheritance between modules, and *refineClass* to refine some part of a class using redefinition of its class members.

Inheritance of modules is expressed as:

$$\begin{aligned} &extends\ :\ \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{M} \\ &extends = \lambda m_t.\ \lambda m_s.\ m_t \rhd m_s \end{aligned}$$

This combinator is the same as that expressing inheritance between outer classes (*extendEncapsulated*) for gbeta.

Classes are refined using *refineClass*:

$$\begin{aligned} &refineClass\ :\ \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{D} \rightarrow \mathcal{M} \\ &refineClass = \lambda m_t.\ \lambda c.\ \lambda d.\ \lambda\epsilon.\ (m_t \backslash c)\epsilon\ \|\{c \mapsto d \rhd (m_t\ \epsilon\ c)\} \end{aligned}$$

MixJuice does not allow multiple versions of a given class to be present at the same time in the same running system. This is the major difference with classboxes apart from the import relationship being specified as inheritance between modules. As a consequence, a program can be composed either of colorless points (*i.e.,* using the point module) or of colored points (*i.e.,* using the coloredPoint module). But a colorless point and a colored point cannot coexist in the same system. Note that this restriction is not expressed in the combinators described above because the calculus is not intended to express program execution.

11 Module System Analysis

We present some of the key characteristics that the different module systems exhibit and that the calculus helps to clearly identify. Figure 7 presents a classification of the module systems we have discussed. The module systems are classified according to properties that enable extension.

11.1 Unextensible Classes

With Java or C#, classes can only be refined through subclassing. The definition of the imported class cannot be enlarged with a set of new definitions (method or field addition or method change) from a package other than the package defining it. We refer to such imported classes as *unextensible*. These extensions have to be defined on subclasses.

This restriction is expressed within the calculus by (i) *fixing* the module from which the imported class comes from and (ii) not extending this class with $\rhd$ (override) or $\|$ (extend). Basically, this is identified by (i) the pattern $(fix\ m_s)\ c$ present in the import statement and by (ii) the absence of $\rhd$ or $\|$.

For instance, importing a class in Java (Section 3) is defined as:

$$importClass = \lambda m_t.\ \lambda m_s.\ \lambda c.\ (hide\ c)(m_t\ \|\ \{c \mapsto (fix\ m_s)\ c\})$$

The new class c is defined as $((fix\ m_s)\ c)$ which makes it unextensible because it contains *fix* and no overriding or extension operations.

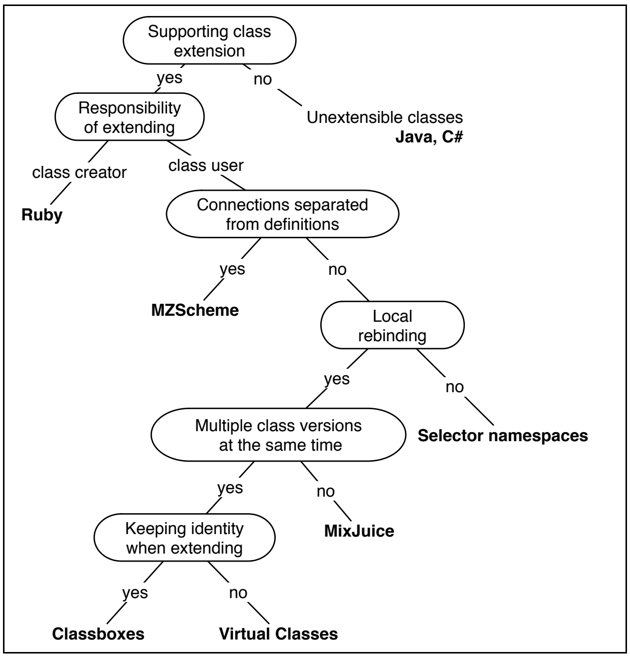

Figure 7: Taxonomy of different module systems.

11.2 Class Extensions

Some module systems allow methods to be added or redefined separately from the definition of the class they belong to. The specification of the class is therefore spread over more than one module. This mechanism complements subclassing. A class extension is the result of a separation between a definition of a class and definitions that compose this class (*i.e.,* method definitions). AspectJ [5] with the notion of inter-type declaration, MultiJava [35] with open-classes, HyperJ [41] with hyper-slices, Smalltalk, CLOS, and Objective-C offer such a mechanism. Within such systems, there is a conceptual difference between a class definition and its method definitions: a method definition is not physically included in a class definition but can be defined externally to the class it belongs to.

The ability of a module system to offer class extensions is expressed in applying an $\triangleright$ or a $\|$ operator to the imported class. For instance, ModularSmalltalk (Section 6) and classboxes (Section 8) allow a class to be extended by adding or redefining methods. This is illustrated in the *extend* combinator for selector namespaces (Section 6):

$$extend = \lambda m_t.\ \lambda m_s.\ \lambda c.\ \lambda d.\ m_t \| \{c \mapsto d \triangleright ((\mathit{fix}\ m_s)\ c)\}$$

The imported class c is the result of $((\mathit{fix}\ m_s)\ c)$, *i.e.,* the definition of the class c looked up in the fixed (self-rebound) module. A set of new definitions is added to it by using $d \rhd \_$. The class obtained from the provider module m_s is extended with a set of definitions d.

With virtual classes, class extensions and mixins, a part of the definitions composing a class can be stated at a different location than the class declaration. Gbeta allows a class to be refined in another unit of modularization (*i.e.,* outer classes). With selector namespaces, part of the behavior can be defined in a namespace different from the one where the class is declared.

Aspect languages offer certain features similar to those offered by module systems. We do not consider these in our comparison, however, for the simple reason that aspects refer to runtime semantics in the specification of pointcuts. This is out of the scope of the present work.

11.3 Extension Responsibility

We defined a class extension as a method addition or redefinition for an already existing class (Section 6). The responsibility of extending a class belongs to its users. For instance, with selector namespaces or classboxes, a class is imported, and then extended. This responsibility belongs to user of this class, and not to its creator. With classboxes this is expressed with the *extend* combinator intended to be applied to an already existing class (Section 8).

$$extend = \lambda m_t.\ \lambda m_s.\ \lambda c.\ \lambda d.\ \lambda \epsilon.\ m_t\ \epsilon \| \{c^{m_s} \mapsto d \rhd\ (m_s\ \epsilon\ \ c)\}$$

The $(m_s\ \epsilon\ \ c)$ contained in *extend* assumes that the class already exists in m_s. The variable d contains features used to extend the class c defined in m_s with.

The decision to use a mixin or not when creating a class is made by the creator of the class. This is why, in Ruby, including a module mixin in the definition of a class is not a class extension in the sense we defined previously. The choice of using a mixin is taken when the class is created. This is expressed with the *newClassWithMixin* combinator where the mixin to use is designated when the class is created.

$$\begin{aligned} newClassWithMixin = {} & \lambda m_t.\ \lambda mixin.\ \lambda c.\ \lambda d. \\ & m_t \| \{c \mapsto fix(\lambda\sigma.\ d \rhd mixin\ \sigma)\} \end{aligned}$$

No previous class definition is looked up in a provider module. The variable d contains the features that the new class c will be composed of, and *mixin* contains the mixin module used by c.

11.4 Local Rebinding

The local rebinding property is provided by an extension mechanism that enables extensions and former definitions of the code to be visible to each other. A change introduced by a class extension can invoke the former definitions, and former definitions can also invoke the new extension.

A module system offers a local rebinding property if within an *import* statement a *fix* operator is *not* applied to the module from which a class is imported. The effect of this fixpoint is to make the imported class see definitions of the importing module. In the studied module systems, gbeta, classboxes, and MixJuice have the local rebinding property but not selector namespaces.

For instance, in gbeta (Section 7), refinements over a set of inner classes are defined within a subclass of the encapsulating class using the *extendEncapsulated* combinator:

$$extendEncapsulated = \lambda m_t.\ \lambda m_s.\ m_t\ \rhd m_s$$

As no *fix* operator is involved, the class definition in the parent encapsulating class can introduce further refinements.

In the same way the *extend* combinator used to express the class extension with classboxes is:

$$extend = \lambda m_t.\ \lambda m_s.\ \lambda c.\ \lambda d.\ \lambda \epsilon.\ m_t\ \epsilon \| \{c^{m_s} \mapsto d \rhd\ (m_s\ \epsilon\ \ c)\}$$

The extended class is the value given by $d\rhd\ m_s\ \epsilon\ c$. The extensions d override the definition of c obtained from $m_s\ s$ (the module from which the class is obtained). Methods originally defined in $m_s\ s\ c$ can call methods defined in d.

Selector namespaces (Section 6) allow a class to be imported and then extended with new methods. These new methods can invoke the former methods. However, the other direction is not possible: former methods cannot invoke redefined definitions. We call this property *non-reentrance*. Selector namespaces do not support local rebinding because they are not reentrant. For instance, in Smallscript [39] a German translation could be defined as shown in Figure 8. A namespace German extends the class Object with a German translation of asString. However, the English version of this method belongs to the namespace English where printOn(aStream) is specified. Therefore, the English version of asString is always invoked by printOn(aStream) even if the execution occurs from within the German namespace.

The *extend* combinator used to express the class extension semantics with selectors namespace is:

$$extend = \lambda m_t.\ \lambda m_s.\ \lambda c.\ \lambda d.\ m_t \| \{c \mapsto d \rhd ((\mathit{fix}\ m_s)\ c)\}$$

The module m_s is fixed before taking the definition of the class c intended to be extended. The definition obtained from $((\mathit{fix}\ m_s)\ c)$ cannot call the extensions defined by d.

With virtual classes or classboxes, on the other hand, a German translation would be printed whenever the printOn(aStream) is invoked from within the

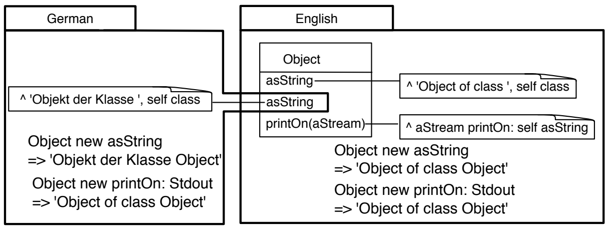

Figure 8: In Modular Smalltalk, namespace selectors are not reentrant: invoking printOn() from the German namespace does not invoke the German version of asString.

package German. These two systems exhibit the local rebinding property. Local rebinding is characterized by taking into account the calling context.

11.5 Privacy in a Module

Module privacy is expressed using the *hide* operator. For example to declare a class as private within a Java package, one may use:

$$private = \lambda m.\ \lambda c.\ (hide\ c)\ m$$

With Java and C#, an imported class may be referred to *only* within the importing package or namespace. An imported class does not belong to the module's interface, and so, cannot be imported from that module by yet another module. This is expressed by the *hide* operator applied to the result of the import. For instance, the C# *usingClassAs* combinator (Section 4) which expresses class import with an alias, is defined as:

$$usingClassAs = \lambda m_t.\ \lambda m_s.\ \lambda a.\lambda c.\ (hide\ a)(m_t \| \{a \mapsto (fix\ m_s)\ c\})$$

The use of $(hide\ a)$ prevents one from importing the class from another module. An imported class cannot be re-imported by another module. For instance, the following Java code is illegal:

```
package a; class C { };
package b; import a.C;
package c; import b.C; // Error, class C is not accessible from package b
```

On the other hand, with classboxes (Section 8), a class can be imported, extended and then imported again by another classbox. The *extend* combinator used to express extension is:

$$extend = \lambda m_t.\ \lambda m_s.\ \lambda c.\ \lambda d.\ \lambda \epsilon.\ m_t\ \epsilon \| \{c^{m_s} \mapsto d \rhd\ (m_s\ \epsilon\ \ c)\}$$

A class that is imported by a classbox and extended is available to clients of that classbox.

11.6 Mixin Behavior

A Ruby module may define functions that are turned into methods whenever they are used by a class (Section 5). Such a behavior is called a module mixin in the Ruby community. The module mixin is applied to the environment representing the class being defined. This is expressed by the use of the fixpoint operator *fix* within *newClassWithMixin*.

$$\begin{aligned} newClassWithMixin = \lambda m_t.\ \lambda mixin.\ \lambda c.\ \lambda d. \\ m_t \| \{c \mapsto fix(\lambda\sigma.\ d \rhd mixin\ \sigma)\} \end{aligned}$$

The expression $mixin\ \sigma$ binds the class being defined σ to the module argument ϵ of $mixin$. This mechanism is illustrated here with Ruby's module mixin but it is also applicable to other mixin mechanisms like those of MzScheme [24] or Jigsaw [9].

11.7 Identity of the Extended Classes

Refining a class by subclassing it does not preserve class identity: the original and refined definitions are implemented by two distinct classes. With virtual classes (Section 7), a class is refined by creating a new class that substitutes the first one when a class lookup is performed. As a consequence, an instance of a class is not an instance of the refined class.

With classboxes, new methods can be added or redefined on an imported class. The new methods are part of the class behavior but they are only visible from the context of the classbox that defines them. As a consequence, these methods are only accessible in this classbox and in other classboxes that import the class from the extending classbox. The identity of the class is preserved. As a consequence, the set of methods understandable by an instance of a class created by a classbox $CB1$ may be enlarged if this instance is referenced by some code in a classbox $CB2$. This is expressed in our calculus by making the originating classbox explicit by means of a superscript (*e.g.,* c^{m_s}).

11.8 Multiple Class Versions at the Same Time

A class defined in a classbox can be refined in another without conflicting with the original definition. This is a result of allowing multiple versions of the same class to coexist in the same system. Each version of a given class can have different collaborating classes present in the same system.

MixJuice offers the possibility of extending a system by defining differential modules. Such modules are then composed to form a executable system. However, one strong constraint is that only one particular version of a class can be present in a system. Therefore, a particular combination of modules may lead to some unexpected results because some modifications might be propagated to clients that rely on the original version only.

One current limitation of our formalism is that the restriction of having only one particular version of a class present in a system is not reflected by the combinators described above. The notion of executable system is not defined, therefore no restriction related to the execution can be formulated.

11.9 Connections Separated from Module Definitions

The advantage of units over conventional module and class languages is that connections between modules or classes are separately specified from their definitions. Separating the definition of classes from their use in different modules makes it easy to replace the original classes with new classes without modifying the client.

12 Related Work

Considerable effort has been invested in studying theoretical foundations of module systems, but to the best of our knowledge there is no work defining a calculus to compare existing object-oriented module systems. We limit this section to summarizing work done in expressing module systems of various object-oriented languages.

In their work on mixins operators Van Limberghen and Mens [32] present the operator *encaps* appropriated to deal with multiple inheritance problems, which is an alternative to the *hide* operator proposed by Bracha and Lindstrom [12]. They mainly focused on multi-inheritance mechanisms.

Ancona and Zucca [2] define a module calculus suitable for encoding various existing mechanism for composing modules. They define a module as a set of imports, a set of exports, and a set of function definitions, *i.e.,* components. Modules are composed using a set of operators: *sum*, *reduct*, *freeze*, *selection.* The operator *sum* glues two modules together, and is roughly equivalent to our *override* (in our calculus, import and export are not explicitly part of a module). The operator *reduct* is a form of renaming; import and export components are separately renamed via two renamings. The *freeze* operators binds input to output names. Finally, *selection* is used by clients of a module to access its components. Their approach enables a large variety of existing mechanisms for combining software components to be expressed (*e.g.,* ML functions, mixin modules). However, no attempt has been made to express module systems of mainstream object-oriented languages.

Bono *et al.,* [8], define some basic object-oriented constructs in a lambda-calculus with records. While they focus on expressing mixin composition as the primary extension mechanism, they do not address the notion of modules and composition operators.

Leroy [31] presents an implementation of an SML-like module system. The SML module system consists of three notions: a structure which is a set of named components, a signature which is an interface for a structure, and a functor which is a function that maps a structure into a new structure. A module is defined by a structure which can be associated with more than one signature. A module can either be user-defined, or the result of applying a functor to another module. Leroy describes an attempt at transferring thus module system to other languages such as *core C* and *mini-ML* which are subsets of C and of ML, respectively.

Linking modules together by functor application prevents the definition of mutually recursive types or procedures across modules boundaries [23]. Objective Caml [38] provides an object-oriented layer as well as an SML-like module system. We did not include this in our comparison because it would be redundant with the study of MZScheme units.

13 Conclusion

We have defined a simple calculus in which modules and classes are combined using a set of basic operators like *hide* and *fix*. Then, for various object-oriented programing languages, we expressed their module systems (*i.e.,* Java packages, C# namespaces, gbeta virtual classes, ...) by defining combinators like *import* or *extend*. The focus of this work is to express various packaging mechanisms using a common foundation. Results of this analysis are summarized in the taxonomy presented in Section 11. Even if only a very few languages are treated in this paper compared to the number of module systems proposed over the last decades, mainstream languages as well as representative languages are studied.

When defining the representation of classes, we expressed inheritance with the *extendClass* combinator. However, we did not model the *super* reference. This would have introduced additional complexity to the calculus that would have shifted the focus of this paper. Furthermore, not all the of languages we considered support *super* (*e.g.,* gbeta).

The presented calculus is untyped. As a future work we plan to explore typing rules for this calculus in order to express, for instance, which compositions of modules are type safe. Virtual classes (like in Eiffel) represent an unchecked use of covariance, which is not type-safe [13], whereas the gbeta approach was always based on checking for covariance (which is possible because, in contrast to Eiffel, covariance is always explicitly declared).

In this paper we mainly focused on expressing import and extend relationship. Our future work is to apply our approach to other systems such as Modula-3 [16], ModularJava [18], JavaMod [3] and Nested Inheritance [37] as they offer the notion of explicit interface.

Numerous formalisms have been developed in recent years to model new kinds of module systems and their features. However, to our knowledge, ours is the first attempt to develop a general calculus for modeling and comparing the diversity of module systems provided by various mainstream object-oriented programming languages.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "A Unified Approach to Composition and Extensibility" (SNF Project No. 200020-105091/1, Oct. 2004 - Sept. 2006) and "Recast: Evolution of Object-Oriented Applications" (SNF 2000-061655.00/1).

We would like to thank Gilad Bracha, William Cook, Erik Ernst, and Eric Tanter for the valuable discussions we have had, which helped to improve the presentation of this paper.

References

1. D. Ancona and E. Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, August 1998.
2. D. Ancona and E. Zucca. A primitive calculus for module systems. In G. Nadathur, editor, *Principles and Practice of Declarative Programming*, number 1702 in LNCS, pages 62–79. Springer Verlag, 1999.
3. D. Ancona and E. Zucca. True modules for Java-like languages. In J. L. Knudsen, editor, *ECOOP 2001*, number 2072 in LNCS, pages 354–380. Springer Verlag, 2001.
4. Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley, 1996.
5. AspectJ home page. http://eclipse.org/aspectj/.
6. Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems and Structures*, 31(3-4):107–126, May 2005.
7. Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of JMLC 2003 (Joint Modular Languages Conference)*, volume 2789 of *LNCS*, pages 122–131. Springer-Verlag, 2003.
8. Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In R. Guerraoui, editor, *Proceedings ECOOP '99*, volume 1628 of *LNCS*, pages 43–66, Lisbon, Portugal, June 1999. Springer-Verlag.

9. Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, Dept. of Computer Science, University of Utah, March 1992.
10. Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, October 1990.
11. Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. Uucs-91-017, University of Utah, Dept. Comp. Sci., October 1991.
12. Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 282–290, April 1992.
13. Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *Proceedings ECOOP '98*, pages 523–549. Springer-Verlag, 1998.
14. C#. http://www.ecma-international.org/publications/standards/Ecma-334.htm.
15. Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.
16. Luca Cardelli, Jim Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, August 1992.
17. Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145, 2000.
18. John Corwin, David F. Bacon, David Grove, and Chet Murthy. MJ: a rational module system for Java and its applications. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 241–254. ACM Press, 2003.
19. Linda G. DeMichiel and Richard P. Gabriel. The common lisp object system: An overview. In J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP '87*, volume 276 of *LNCS*, pages 151–170, Paris, France, June 1987. Springer-Verlag.
20. DrScheme. http://www.drscheme.org/.
21. Erik Ernst. Propagating class and method combination. In R. Guerraoui, editor, *Proceedings ECOOP '99*, volume 1628 of *LNCS*, pages 67–91, Lisbon, Portugal, June 1999. Springer-Verlag.
22. Erik Ernst. Family polymorphism. In J. L. Knudsen, editor, *ECOOP 2001*, number 2072 in LNCS, pages 303–326. Springer Verlag, 2001.
23. Matthew Flatt and Matthias Felleisen. Units: Cool modules for hot languages. In *Proceedings of PLDI '98 Conference on Programming Language Design and Implementation*, pages 236–248. ACM Press, 1998.
24. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, 1998.
25. Yuuji Ichisugi and Akira Tanaka. Difference-based modules: A class independent module mechanism. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, Malaga, Spain, June 2002. Springer Verlag.
26. Atsushi Igarashi and Benjamin C. Pierce. On inner classes. *Lecture Notes in Computer Science*, 1850, 2000.
27. Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 318–326. ACM Press, November 1997.
28. Java. http://java.sun.com/.
29. Sonia E. Keene. *Object-Oriented Programming in Common-Lisp.* Addison Wesley, 1989.

30. Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. The BETA programming language. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. MIT Press, Cambridge, Mass., 1987.
31. Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.
32. Tom Mens and Marc van Limberghen. Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.
33. Bertrand Meyer. *Eiffel: The Language.* Prentice-Hall, 1992.
34. Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99. ACM Press, 2003.
35. Todd Millstein, Mark Reay, and Craig Chambers. Relaxed multijava: balancing extensibility and modular typechecking. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 224–240. ACM Press, 2003.
36. Hanspeter Mössenböck. *Object-Oriented Programming in Oberon-2.* Springer-Verlag, 1993.
37. Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 99–115. ACM Press, 2004.
38. Ocaml. http://caml.inria.fr/.
39. Dave Simmons. Smallscript, 2002. http://www.smallscript.com.
40. S. Tucker Taft. Ada 9x: From abstraction-oriented to object-oriented. In *Proceedings OOPSLA '93*, volume 28, pages 127–143, October 1993.
41. Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proceedings of ICSE '99*, pages 107–119, Los Angeles CA, USA, 1999.
42. David Thomas and Andrew Hunt. *Programming Ruby.* Addison Wesley, 2001.
43. Cincom Smalltalk, September 2003. http://www.cincom.com/scripts/smalltalk.dll/.
44. Allen Wirfs-Brock and Brian Wilkerson. An overview of modular Smalltalk. In *Proceedings OOPSLA '88*, pages 123–134, November 1988.
45. Matthias Zenger. Type-safe prototype-based component evolution. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, pages 470–497, Malaga, Spain, June 2002. Springer Verlag.