# A Core Calculus of Classes and Mixins

Viviana Bono,[1] Amit Patel,[2] and Vitaly Shmatikov[2]

[1] Dipartimento di Informatica dell'Università di Torino, C.so Svizzera 185, 10149 Torino, Italy, bono@di.unito.it
(currently at the School of Computer Science, The University of Birmingham, Birmingham B15 2TT, United Kingdom, v.bono@cs.bham.ac.uk)
[2] Computer Science Department, Stanford University, Stanford, CA 94305-9045, U.S.A., {amitp,shmat}@cs.stanford.edu

**Abstract.** We develop an imperative calculus that provides a formal model for both single and mixin inheritance. By introducing classes and mixins as the basic object-oriented constructs in a $\lambda$-calculus with records and references, we obtain a system with an intuitive operational semantics. New classes are produced by applying mixins to superclasses. Objects are represented by records and produced by instantiating classes. The type system for objects uses only functional, record, and reference types, and there is a clean separation between subtyping and inheritance.

**Keywords:** object-oriented language, mixin, class, inheritance, calculus, operational semantics, type system.

## 1 Introduction

Mixins (classes parameterized over superclasses) have become a focus of active research both in the software engineering [43, 41, 25] and programming language design [11, 12, 10, 35, 30] communities. Mixin inheritance has been shown to be an expressive alternative to multiple inheritance and a powerful tool for implementing reusable class hierarchies. However, there has been a dearth of formal calculi to provide a theoretical foundation for mixin inheritance and, in particular, few attempts have been made to use mixins as the basic inheritance construct in the core calculus. Although mixin inheritance is easy to formalize in an untyped setting, static type checking of mixins at the time of declaration (as opposed to the time of mixin use) is more difficult. In addition, many approaches to mixins do not address the modular construction of objects, including initialization of fields.

While popular object-oriented languages such as C++ [42] and JAVA [3] are overwhelmingly class-based, most previous core calculi for object-oriented languages were based on objects. In our framework, classes and mixins are basic constructs. The decision to directly include classes in a core calculus reflects many years of struggle with object-based calculi. In simple terms, there is a fundamental conflict between inheritance and subtyping of object types [20, 14, 6, 28]. Our calculus resolves this conflict by supporting class inheritance without class subtyping and object subtyping without object extension. The separation between inheritance (an operation associated with classes) and run-time manipulation of objects allows us to represent objects by records and keep the type system for objects simple, involving only functional, record, and reference types. In particular, we do not need polymorphic object types or recursive *MyType*.

An important advantage of our type system is that it gives types to mixin declarations and mixin applications separately. The actual class to which the mixin is applied may have a "richer" type than that expected by the mixin. For example, it may have more methods, or the types of its methods may be subtypes of those assumed when typing the mixin. This facilitates modular development of class hierarchies, promotes reuse of mixins, and enables the programmer to use a single mixin to add the same functionality to a wide variety of classes. Name clashes between mixins and classes to which they are applied are detected and resolved at the time of mixin application.

We discuss design motivations and tradeoffs, and give a brief overview of the core calculus in section 2. We then present the syntax of the calculus (section 3), its operational semantics (section 4), and the type system (section 5). Finally, we compare our calculus with other object-oriented calculi and indicate directions for future research.

A simpler version of the calculus described in this paper was presented at MFPS '99 [7]. The calculus of [7] supports conventional single inheritance instead of mixins.

## 2 Design of the Core Calculus

In this section, we present our design motivations, discuss tradeoffs involved to designing calculi for object-oriented languages, give a short overview of our calculus, and present an example illustrating mixin usage.

### 2.1 Design Motivations

Our goal is to design a simple class-based calculus that correctly models the basic features of popular class-based languages and reflects modular programming techniques commonly employed by working programmers. Modular program development in a class-based language involves minimizing code dependencies such as those between a superclass and its subclasses, and between a class implementation and object users. Our calculus minimizes dependencies by directly supporting data encapsulation, mixin inheritance, structural subtyping, and modular object creation.

Data encapsulation. We use the C++ terminology (private, protected, and public) for levels of encapsulation. Unlike C++ and some approaches to encapsulation in object calculi such as existential types, our levels of encapsulation describe *visibility*, and not merely *accessibility*. For example, in our calculus even the names of private items are invisible outside the class in which they are defined.

We believe that this is a better approach since *no* information about data representation is revealed — not even the number and names of fields. One of the benefits of using visibility-based encapsulation is that no conflicts arise if both the superclass and the subclass declare a private field of the same name. Among other advantages, this allows the same mixin to be applied twice (see the example in section 2.4).

Mixin inheritance. A *mixin* is a class definition parameterized over the superclass. The decomposition of ordinary inheritance into mixins plus mixin application is similar to the decomposition of let binding into functions plus function application. A mixin can be viewed as a function that takes a class and derives a new subclass from it. The same mixin can be applied to many classes, obtaining a family of subclasses with the same set of methods added and/or replaced. By providing an abstraction mechanism for inheritance, mixins remove the dependency of the subclass on the superclass, enabling modular development of class hierarchies — *e.g.*, a subclass can be implemented before its superclass has been implemented. Mixin inheritance can be used to model single inheritance and many common forms of multiple inheritance [11, 9].

Mixins were first introduced in the Flavors system [38] and CLOS [33], although as a programming idiom rather than a formal language construct. Our calculus is an attempt to formalize mixins as the basic mechanism underlying all inheritance. To ensure that mixin inheritance can be statically type checked, our calculus employs constrained parameterization. From each mixin definition the type system infers a constraint specifying to which classes the mixin may be applied so that the resulting subclass is type-safe. The constraint includes both positive (which methods the class must contain) and negative (which methods the class may not contain) information. The actual class to which the mixin is applied does not have to match the constraint exactly. It may have more methods than required by the positive part of the constraint, and the types of the required methods may be different from those specified by the constraint as long as the resulting subclass is type-safe.

We believe that new and redefined methods should be distinguished in the mixin implementation. From the implementor's viewpoint, a new method may have arbitrary behavior, while the behavior of a redefined method must be "compatible" with that of the old method it replaces. Having this distinction in the syntax of our calculus helps mixin implementors avoid unintentional redefinitions of superclass methods and facilitates generation of the mixin's superclass constraint (see section 4). It also helps resolve name clashes when the mixin is applied. Suppose the mixin and the class to which it is applied both define a method with the same name. If the mixin method is marked as *redefined*, then it is put in the resulting subclass (subject to type compatibility with the replaced method). If the mixin method is marked as *new*, the type system signals an error.

Structural subtyping. As in most popular object-oriented languages, objects in our calculus can only be created by instantiating a class. In contrast to C++, where an object's type is related to the class from which it was instantiated and subtyping relations apply only to object instantiated from the same class hierarchy, we made a deliberate design decision to use *structural subtyping* in order to remove the dependency of object users on class implementation. Objects created from unrelated classes can be substituted for each other if their types satisfy the subtyping relation.

Modular object construction. Class hierarchies in a well-designed object-oriented program must not be fragile: if a superclass implementation changes but the specification remains intact, the implementors of subclasses should not have to rewrite subclass implementations. This is only possible if object creation is modular. In particular, a subclass implementation should not be responsible for initializing inherited fields when a new object is created, since some of the inherited fields may be private and thus invisible to the subclass. Also, the definitions of inherited fields may change when the class hierarchy changes, making the subclass implementation invalid. Instead, the object construction system should call a class *constructor* to provide initial values only for that class's fields, call the superclass constructor to provide initial values for the superclass fields, and so on for each ancestor class. This approach is used in many object-oriented programming languages, including C++ and JAVA.

Unlike many theoretical calculi for object-oriented languages, our calculus directly supports modular object construction. The mixin implementor only writes the local constructor for his own mixin. Mixin applications are reduced to generator functions which call all constructors in the inheritance chain in the correct order, producing a fully initialized object (see section 4).

### 2.2 Design Tradeoffs

In this section, we explain the design decisions and tradeoffs chosen in our calculus. Our goal was to sacrifice as little expressive power as possible while keeping the type system simple and free of complicated types such as polymorphic object types and recursive *MyType*.

Classes. Even in purely object-based calculi, the conflict between inheritance and subtyping usually requires that two sorts of objects be distinguished [28]. "Prototype objects" do not support full subtyping but can be extended with new methods and fields and/or have their methods redefined. "Proper objects" support both depth and width subtyping but are not extensible. Without this distinction, special types with extra information are required to avoid adding a method to an object in which a method with the same name is hidden as a consequence of subtyping (*e.g.*, labeled types of [6]). In our calculus, the class construct plays the role of a "prototype" (extensible but not subtypable), while objects — represented by records of methods — are subtypable but not extensible.

Objects. Records are an intuitive way to model objects since both are collections of name/value pairs. The records-as-objects approach was in fact developed in the pioneering work on object-oriented calculi [19], in which inheritance was modeled by record subtyping. Unlike records, however, object methods should be able to modify fields and invoke sibling methods [21]. To be capable of updating the object's internal state, methods must be functions of the host object (*self*). Therefore, objects must be *recursive* records. Moreover, *self* must be appropriately updated when a method is inherited, since new methods and fields may have been added and/or old ones redefined in the new host object. In our calculus, reduction rules produce class generators that are carefully designed so that methods are given a (recursive) reference to *self* only after inheritance has been resolved and all methods and fields contained in the host object are known.

Object updates. If all object updates are imperative, *self* can be bound to the host object when the object is instantiated from the class. We refer to this approach as *early self* binding. Self then always refers to the same record, which is modified imperatively in place by the object's methods. The main advantage of early binding is that the fixed-point operator (which gives the object's methods reference to *self*) has to be applied only once, at the time of object instantiation.

If functional updates must be supported — which is, obviously, the case for purely functional object calculi — early binding does not work (see, for example, [1], where early binding is called *recursive semantics*). With functional updates, each change in the object's state creates a new object. If *self* in methods is bound just once, at the time of object instantiation, it will refer to the old, incorrect object and not to the new, updated one. Therefore, *self* has to be bound each time a method is invoked. We refer to this approach as *late self* binding.

Object extension. Object extension in an object-based calculus is typically modeled by an operation that extends objects by adding new methods to them. There are two constraints on such an operation: (i) the type system must prevent addition of a method to an object which already contains a method with the same name, and (ii) since an object may be extended again after method addition, the actual host object may be larger than the object to which the method was originally added. The method body must behave correctly in any extension of the original host object. Therefore, it must have a polymorphic type with respect to *self*. The fulfillment of the two constraints can be achieved, for instance, via polymorphic types built on row schemes [5] that use kinds to keep track of methods' presence.

Even more complicated is the case when object extension must be supported in a functional calculus. In the functional case, all methods modifying an object have the type of *self* as their return type. Whenever an object is extended or has its methods redefined (overriden), the type given to *self* in all inherited methods must be updated to take into account new and/or redefined methods. Therefore, the type system should include the notion of *MyType* (a.k.a. *SelfType*) so that the inherited methods can be specialized properly. Support for *MyType* generally leads to more complicated type systems, in which forms of recursive types are required. Support can be provided by using row variables combined with recursive types [27, 26, 28], match-bound type variables [18, 4], or by means of special forms of second-order quantifiers such as the *Self* quantifier of [1].

Tradeoffs. Our goal is to achieve a reasonable tradeoff between expressivity and simplicity. We do not support functional updates because we believe that imperative updates combined with early *self* binding provide such a tradeoff. Without functional updates, we can use early binding of *self*. Early binding eliminates the main need for recursive object types. There is also no need for polymorphic object types in our calculus since inheritance is modeled entirely at the class level and there are no object extension operations. This choice allows us to have a simple type system and a straightforward form of structural subtyping, in contrast with the calculi that support *MyType* specialization [28, 18].

There are at least two possible drawbacks to our approach. Although methods that *return* a modified *self* can be modeled in our calculus as imperative methods that modify the object and return nothing, methods that accept a *MyType argument* cannot be simulated in our system without support for *MyType*. We therefore have no support for binary methods of the form described in [15]. Also, the type system of our calculus does not directly support *implementation types* (*i.e.*, types that include information about the class from which the object was instantiated and not just the object's interface). We believe that a form of implementation types can be provided by extending our type system with existential types.

### 2.3 Design of the Core Calculus

The two main concepts in object-oriented programming are *objects* and *classes*. In our calculus, objects are records of methods. Methods are represented as functions with a binding for *self* (the host record) and *field* (the private field). Since records, functions, and $\lambda$-binding are standard, we need not introduce new operational semantics or type rules for objects. Instead, we introduce new constructs and rules for mixins and classes only. The new constructs are: *class values* (representing complete classes obtained as a result of mixin application), *mixin expressions* (containing definitions of methods, fields, and constructors), and *instantiation expressions* (representing creation of objects from classes).

A class value is a tuple containing the generator function, the set of all method names, and the set of method names that are protected. The generator produces a function from *self* to a record of methods. When the class is instantiated, the fixed-point operator is applied to the generator's result to bind *self* in the methods' bodies, creating a full-fledged object.

Mixins — *i.e.*, classes parameterized over the superclass — are represented by mixin expressions. Inheritance is modeled by the evaluation rule that applies a mixin to a class value representing the superclass, producing a new class value. The generator of the new class takes the record of superclass methods built by the superclass generator and modifies it by adding and/or replacing methods as specified by the mixin. Only class values can be instantiated; mixins are used solely for building class hierarchies.

For simplicity, the core calculus supports only private fields and public and protected methods. Private methods can be modeled by private fields with a function type; public or protected fields can be modeled by combining private fields with accessor methods. Instead of putting encapsulation levels into object types, we express them using subtyping and binding. Protected methods are treated in the same way as public methods except that they are excluded from the type of the object returned to the user. Private fields are not listed in the object type at all, but are instead bound in each method body. In the core calculus each class has exactly one private field, which may have a record type. Each method body takes the class's private field as a parameter.

### 2.4 An Example of Mixin Inheritance

Mixin inheritance can be a powerful tool for constructing class hierarchies. In this section, we give a simple example that demonstrates how a mixin can be implemented in our calculus and explain some of the uses of mixins. For readability, the example uses functions with multiple arguments even though they are not formalized explicitly in the calculus.

Mixin definition. Following is the definition of Encrypted mixin that implements encryption functionality on top of any stream class. Note that the class to which the mixin is applied may have more methods than expected by the mixin. For example, Encrypted can be applied to Socket $\diamond$ *Object* where *Object* is the root of all class hierarchies, even though Socket $\diamond$ *Object* has other methods besides *read* and *write*.

```
let File =
  mixin
    method write = ...
    method read = ...
    ...
  end in

let Socket =
  mixin
    method write = ...
    method read = ...
    method hostname = ...
    method portnumber = ...
    ...
  end in

let Encrypted =
  mixin
    redefine write = λnext. λkey. λself. λdata. next (encrypt(data, key));
    redefine read =  λnext. λkey. λself. λ_ .   decrypt(next (), key);
    constructor λ(key, arg). {fieldinit=key, superinit=arg};
    protect [];
  end in ...
```

Mixin expressions contain new methods (marked by the method keyword), redefined methods (redefine keyword), and constructors. The names of protected methods should be listed following the protect keyword. Instead of introducing a special field construct, every mixin contains a single private field which is $\lambda$-bound in each method body ($\lambda$key. $\cdots$).

Methods can access the host object through the *self* parameter, which is $\lambda$-bound in each method body to avoid introducing special keywords. Redefined methods can access the old method body inherited from the superclass via the *next* parameter. Constructors are simply functions returning a record of two components. The *fieldinit* value is used to initialize the private field. The *superinit* value is passed as an argument to the superclass constructor.

From the definition of Encrypted, the type system infers the constraint that must be satisfied by any class to which Encrypted is applied. The class must contain *write* and *read* methods whose types must be supertypes of those given to *write* and *read*, respectively, in the definition of Encrypted.

Mixin usage. To create an encrypted stream class, one must apply the Encrypted mixin to an existing stream class. In our calculus, the notation for applying mixin $M$ to class $C$ is $M \diamond C$. For example, Encrypted $\diamond$ FileStream is an encrypted file class. The power of mixins can be seen when we apply Encrypted to a family of different streams. For example, we can construct Encrypted $\diamond$ NetworkStream, which is a class that encrypts data communicated over a network. In addition to single inheritance, we can express many uses of multiple inheritance by applying more than one mixin to a class. For example, PGPSign $\diamond$ UUEncode $\diamond$ Encrypt $\diamond$ Compress $\diamond$ FileStream produces a class of files that are compressed, then encrypted, then uuencoded, then signed. In addition, mixins can be used for forms of inheritance that are not possible in most single and multiple inheritance-based systems. In the above example, the result of applying Encrypted to a stream satisfies the constraint required by Encrypted itself, therefore, we can apply Encrypted more than once: Encrypted $\diamond$ Encrypted $\diamond$ FileStream is a class of files that are encrypted twice. In our system, private fields of classes do not conflict even if they have the same name, so each application of Encrypted can have its own encryption key. Unlike most forms of multiple inheritance, it is easy to specify the *order* and *number* of times the mixins are applied.

A note on an implementation. Our calculus uses structural object types that retain no connection to the class from which the object was instantiated. Since unrelated classes may use different layouts for the method dictionary, the compiler cannot use the object's static type to determine the exact position of a method in the dictionary in order to optimize method lookup as is done in C++. Adding mixins in this environment does not impose an extra overhead.

It is possible to support efficient method lookup by introducing a separate hierarchy of mixin interfaces similar to the one analyzed by Flatt et al. [30] and requiring that the order of methods in a mixin's dictionary match that given in the interface implemented by the mixin. However, a separate interface hierarchy would make the calculus significantly more complicated.

## 3 Syntax of the Core Calculus

The syntax of our calculus is fundamentally class-based. There are four expressions involving classes: classval, mixin, $\diamond$ (mixin application), and new. Class-related expressions and values are treated as any other expression or value in the calculus. They can be passed as arguments, put into data structures, etc. However, class values and object values are not intended to be written directly;

$$
\begin{array}{lll}
\text{Expressions:} & e ::= & \mathit{const} \mid x \mid \lambda x.e \mid e_1\ e_2 \mid \mathit{fix} \mid \mathsf{ref} \mid\ ! \mid\ := \\
& & \mid \{x_i = e_i\}^{i \in I} \mid e.x \mid \mathsf{H}\ h.e \mid \mathsf{new}\ e \\
& & \mid \mathsf{classval}\langle v_g, [m_i]^{i \in \mathit{Meth}}, [p_\ell]^{\ell \in \mathit{Prot}} \rangle \\
& & \mid \mathsf{mixin} \\
& & \quad \mathsf{method}\ m_j = v_{m_j}; \quad {}^{(j \in \mathit{New})} \\
& & \quad \mathsf{redefine}\ m_k = v_{m_k}; \quad {}^{(k \in \mathit{Redef})} \\
& & \quad \mathsf{protect}\ [p_\ell]; \quad {}^{(\ell \in \mathit{Prot})} \\
& & \quad \mathsf{constructor}\ v_c; \\
& & \ \mathsf{end} \\
& & \mid e_1 \ \diamond\ e_2 \\
\\
\text{Values:} & v ::= & \mathit{const} \mid x \mid \lambda x.e \mid \mathit{fix} \mid \mathsf{ref} \mid\ ! \mid\ := \mid\ := v \mid \{x_i = v_i\}^{i \in I} \\
& & \mid \mathsf{classval}\langle v_g, [m_i]^{i \in \mathit{Meth}}, [p_\ell]^{\ell \in \mathit{Prot}} \rangle \\
& & \mid \mathsf{mixin} \\
& & \quad \mathsf{method}\ m_j = v_{m_j}; \quad {}^{(j \in \mathit{New})} \\
& & \quad \mathsf{redefine}\ m_k = v_{m_k}; \quad {}^{(k \in \mathit{Redef})} \\
& & \quad \mathsf{protect}\ [p_\ell]; \quad {}^{(\ell \in \mathit{Prot})} \\
& & \quad \mathsf{constructor}\ v_c; \\
& & \ \mathsf{end}
\end{array}
$$

**Fig. 1.** Syntax of the core calculus

instead, these expression forms are used only to define the semantics of programs. Class values are created by mixin application, and object values are created by class instantiation.

Let *Var* be an enumerable set of variables (otherwise referred to as *identifiers*), and *Const* be a set of constants. Expressions $E$ and values $V$ (with $V \subset E$) of the core calculus are as in Fig.1, where $\mathit{const} \in \mathit{Const}$; $x, x_i, m_i, m_j \in \mathit{Var}$; *fix* is the fixed-point operator; ref, !, $:=$ are operators;[1] $\{x_i = e_i\}^{i \in I}$ is a record; $e.x$ is the record selection operation; $h$ is a set of pairs $h ::= \{\langle x, v\rangle^*\}$ where $x \in \mathit{Var}$ and $v$ is a value (first components of the pairs are all distinct); $[m_i], [p_\ell]$ are sets of identifiers; and $I, J, K, L, \mathit{Meth}, \mathit{Prot}, \mathit{New}, \mathit{Redef} \subset \mathbb{N}$.

Our calculus takes a standard calculus of functions, records, and imperative features and adds new constructs to support classes and mixins. We chose to extend *Reference ML* [45], in which Wright and Felleisen analyze the operational soundness of a version of ML with imperative features. Our calculus does not include let expressions as primitives since we do not need polymorphism to model our objects. We do rely on the Wright-Felleisen idea of *store*, which we call *heap*, in order to evaluate imperative side effects.

The expression $\mathsf{H}\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e$ associates reference variables $x_1, \ldots x_n$ with values $v_1, \ldots, v_n$. H binds $x_1, \ldots x_n$ in $v_1, \ldots, v_n$ and in $e$. The set of pairs

[1] Introducing ref, !, $:=$ as operators rather than standard forms such as ref$e$, !$e$, $:=e_1e_2$, simplifies the definition of evaluation contexts and proofs of properties. As noted in [45], this is just a syntactic convenience, as is the curried version of $:=$.

$h$ in the expression $\mathsf{H}h.e$ represents the *heap*, where the results of evaluating imperative subexpressions of $e$ are stored.

The intuitive meaning of the class-related expressions is as follows:

- $\mathsf{classval}\langle v_g, [m_i]^{i \in Meth}, [p_\ell]^{\ell \in Prot}\rangle$ is a *class value*, *i.e.*, the result of mixin application. It is a triple, containing one function and two sets of variables. The function $v_g$ is the generator for the class. The $[m_i]$ set contains the names of all methods defined in the class, and the $[p_\ell]$ set contains the names of protected methods.
- mixin
  method $m_j = v_{m_j};\quad ^{(j \in New)}$
  redefine $m_k = v_{m_k};\quad ^{(k \in Redef)}$
  protect $[p_\ell];\quad ^{(\ell \in Prot)}$
  constructor $v_c$;
  end
  is a *mixin*, in which $m_j = v_{m_j}$ are definitions of new methods, and $m_k = v_{m_k}$ are method redefinitions that will replace methods with the same name in any class to which the mixin is applied. Each method body $v_{m_{j,k}}$ is a function of *self*, which will be bound to the newly created object at instantiation time, and of the private *field*. In method redefinitions, $v_{m_k}$ is also a function of *next*, which will be bound to the old, redefined method from the superclass. The $v_c$ value in the constructor clause is a function that returns a record of two components. When evaluating a mixin application, $v_c$ is used to build the generator as described in section 4.
- $e_1 \diamond e_2$ is an application of mixin $e_1$ to class value $e_2$. It produces a new class value. Mixin application is the basic inheritance mechanism in our calculus.
- new $e$ uses generator $v_g$ of the class value to which $e$ evaluates to create a function that returns a new object, as described in section 4.

*Programs* and *answers* are defined as follows:

$$p ::= e \qquad \text{where } e \text{ is a closed expression}$$
$$a ::= v \mid \mathsf{H}\ h.v$$

Finally, we define the root of the class hierarchy, class *Object*, as a predefined class value:

$$\mathit{Object} \stackrel{\triangle}{=} \mathsf{classval}\langle\ \lambda\_.\lambda\_.\{\},\ [\ ],\ [\ ]\ \rangle$$

The root class is necessary so that all other classes can be treated uniformly. Intuitively, *Object* is the class whose object instances are empty objects. It is the only class value that is not obtained as a result of mixin application. The calculus can then be simplified by assuming that any user-defined class that does not need a superclass is obtained by applying a mixin containing all of the class's method definitions to *Object*.

Throughout this paper, we will use let $x = e_1$ in $e_2$ in terms and examples as a more readable equivalent of $(\lambda x.e_2)e_1$. Also, we use unit as an abbreviation for the empty record or type $\{\}$, instead of having a new *unit* value and type. We will use the word "object" when the record in question represents an object. To avoid name capture, we apply $\alpha$-conversion to binders $\lambda$ and $\mathsf{H}$.

## 4 Operational Semantics

$$
\begin{array}{rll}
\mathit{const}\ v & \to \delta(\mathit{const}, v) \ \text{ if } \delta(\mathit{const}, v) \text{ is defined} & (\delta)\\
(\lambda x.e)\ v & \to [v/x]\ e & (\beta_v)\\
\mathit{fix}\ (\lambda x.e) & \to [\mathit{fix}(\lambda x.e)/x]e & (\text{fix})\\
\{\ldots, x = v, \ldots\}.x & \to v & (\text{select})\\
\mathsf{ref} v & \to \mathsf{H}\langle x, v\rangle.x & (\text{ref})\\
\mathsf{H}\langle x, v\rangle h.R[!x] & \to \mathsf{H}\langle x, v\rangle h.R[v] & (\text{deref})\\
\mathsf{H}\langle x, v\rangle h.R[:= x v'] & \to \mathsf{H}\langle x, v'\rangle h.R[v'] & (\text{assign})\\
R[\mathsf{H}\ h.e] & \to \mathsf{H}\ h.R[e], \quad R \neq [\,] & (\text{lift})\\
\mathsf{H}\ h.\mathsf{H}\ h'.e & \to \mathsf{H}\ h\ h'.e & (\text{merge})\\
\mathsf{new}\ \mathsf{classval}\langle g, \mathcal{M}, \mathcal{P}\rangle & \to \lambda v.\mathit{Sub}_{\mathcal{M} \to \mathcal{M} \setminus \mathcal{P}}(\mathit{fix}(g\ v)) & (\text{new})
\end{array}
$$

$$
\begin{pmatrix}
\mathsf{mixin} \\
\quad \mathsf{method}\ m_j = v_{m_j}; \\
\quad \mathsf{redefine}\ m_k = v_{m_k}; \\
\quad \mathsf{protect}\ [p_\ell]; \\
\quad \mathsf{constructor}\ c; \\
\mathsf{end}
\end{pmatrix}^{\substack{j \in New, \\ k \in Redef, \\ \ell \in Prot}} \diamond\ \mathsf{classval}\langle g, \mathcal{M}, \mathcal{P}\rangle \to \mathsf{classval}\langle Gen, [m_j] \cup \mathcal{M}, [p_\ell] \cup \mathcal{P}\rangle \qquad (\text{mixin})
$$

if $[m_j] \cap \mathcal{M} = \emptyset$, $[m_k] \subset \mathcal{M}$, and $[p_\ell] \subset [m_j] \cup \mathcal{M}$; $Gen$ is defined below

**Fig. 2.** Reduction Rules

The operational semantics for our calculus extends that of *Reference ML* [45]. Reduction rules are given in Fig.2, where $R$ are *reduction contexts* [22, 24, 37]. Expression *Gen* is defined below. Relation $\twoheadrightarrow$ is the reflexive, transitive, contextual closure of $\to$, with respect to *contexts* $C$, as defined (in a standard way) in appendix A.

*Reduction contexts* are necessary to provide a minimal relative linear order among the creation, dereferencing and updating of heap locations, since side effects need to be evaluated in a deterministic order. Reduction contexts $R$ are defined as follows:

$$
\begin{array}{rl}
R ::= & [\ ] \mid R\ e \mid v\ R \mid R.x \mid \mathsf{new}\ R \mid R\ \diamond\ e \mid v\ \diamond\ R \\
& \mid \{m_1 = v_1, \ldots, m_{i-1} = v_{i-1}, m_i = R, m_{i+1} = e_{i+1}, \ldots, m_n = e_n\}^{1 \leq i \leq n}
\end{array}
$$

To abstract from a precise set of constants, we only assume the existence of a partial function $\delta : \mathit{Const} \times \mathit{ClosedVal} \rightharpoonup \mathit{ClosedVal}$ that interprets the application of functional constants to closed values and yields closed values. See section 5 for the $\delta$-typability condition.

($\beta_v$) and (*select*) rules are standard.

(*ref*), (*deref*) and (*assign*) rules evaluate imperative expressions following the linear order given by the reduction context $R$ and acting on the heap. They are formulated after [45]: (*ref*) generates a new heap location where the value $v$ is stored, (*deref*) retrieves the contents of the location $x$, (*assign*) changes the value stored in a heap location.

(*lift*) and (*merge*) rules combine inner local heaps with outer ones whenever a dereference operator or an assignment operator cannot find the needed location in the closest local heap.

(*mixin*) rule evaluates mixin application expressions which represent inheritance in our calculus. A mixin is applied to a superclass value classval$\langle g, \mathcal{M}, \mathcal{P}\rangle$. $\mathcal{M}$ is a set of all method names defined in the superclass; $\mathcal{P}$ is an annotation listing the names of protected methods in the superclass. The resulting class value is classval$\langle Gen, [m_j] \cup \mathcal{M}, [p_\ell] \cup \mathcal{P}\rangle$ where $Gen$ is the generator function defined below, $[m_j] \cup \mathcal{M}$ is the set of all method names, and $[p_\ell] \cup \mathcal{P}$ is an annotation listing protected method names. Using generators delays full inheritance resolution until object instantiation time when *self* becomes available.

Gen is the class generator. It takes a single argument $x$ which is used by the constructor subexpression $c$ to compute the initial value for the private field of the new object and the argument for the superclass generator. $Gen$ returns a function from *self* to a record of methods. When the fixed-point operator is applied to the function returned by the generator, it produces a recursive record of methods representing a new object (see the (*new*) rule).

$$
\begin{array}{l}
Gen \triangleq \lambda x. \\
\qquad \mathsf{let}\ t = c(x) \quad \mathsf{in} \\
\qquad \mathsf{let}\ supergen = g(t.\text{superinit}) \quad \mathsf{in} \\
\qquad \lambda self. \\
\qquad \quad \left\{ \begin{array}{lll}
m_j = \lambda y.v_{m_j} & & t.\text{fieldinit}\ self\ y \\
m_k = \lambda y.v_{m_k} & (supergen\ self).m_k & t.\text{fieldinit}\ self\ y \\
m_i = \lambda y. & (supergen\ self).m_i & y
\end{array} \right\}^{m_i \in \mathcal{M} \setminus [m_k]}
\end{array}
$$

In the mixin expression, the constructor subexpression $c$ is a function of one argument which returns a record of two components: one is the initialization expression for the field (*fieldinit*), and the other is the superclass generator's argument (*superinit*). $Gen$ first calls $c(x)$ to compute the initial value of the field and the value to be passed to the superclass generator $g$. $Gen$ then calls the superclass generator $g$, passing argument $t$.superinit, to obtain a function *supergen* from *self* to a record of superclass methods.

Finally, $Gen$ builds a function from *self* that returns a record containing *all* methods — from both the mixin and the superclass. To understand how the record is created, recall that method bodies take parameters *field*, *self*, and, if it's a redefinition, *next*. Methods $m_j$ are the *new* mixin methods: they appear for the first time in the current mixin expression. $Gen$ must bind *field* and *self* for them. Methods $m_i \in \mathcal{M} \setminus [m_k]$ are the *inherited* superclass methods: they are taken intact from the superclass's object (*supergen self*). Methods $m_k$ are *redefined* in the mixin. Their bodies can refer to the old methods through the *next* parameter, which is bound to (*supergen self*)$.m_i$ by $Gen$. They also receive a binding for *field* and *self*. For all three sorts of methods, the method bodies are wrapped inside $\lambda y.\cdots y$ to delay evaluation in our call-by-value calculus.

(*fix*) rule is standard.

(*new*) rule builds a function that can create a new object. The resulting function can be thought of as the composition of three functions: $Sub \circ fix \circ g$. Given an argument $v$, it will apply generator $g$ to $v$, creating a function from *self* to a record of methods. Then the fixed-point operator *fix* (following [21]) is applied to bind *self* in method bodies and create a recursive record. Finally, we apply $Sub_{\mathcal{M}\to\mathcal{M}\setminus\mathcal{P}}$, a coercion function from records to records that hides all components belonging to the protected set $\mathcal{P}$. The resulting record contains only public methods, and can be returned to the user as a fully formed object.

## 5 Type System

Our types are standard and the typing rules are fairly straightforward. The complexity of typing object-oriented programs in our system is limited exclusively to classes and mixins. Method selection, which is the only operation on objects in our calculus, is typed as ordinary record component selection. Since methods are typed as ordinary functions, method invocation is simply a function application.

Types are as follows:

$$\begin{aligned}\tau ::= {} & \iota \mid \tau_1 \to \tau_2 \mid \tau\ \mathsf{ref} \mid \{x_i : \tau_i\}^{i\in I} \\ & \mid \mathsf{class}\langle \tau, \{m_i{:}\tau_i\}, [p_\ell]\rangle^{i\in I, \ell\in L} \\ & \mid \mathsf{mixin}\langle \tau_1, \tau_2, \{m_j{:}\tau_j\}, \{m_k{:}\tau_k\}, [p_\ell]\rangle^{j\in J, k\in K, \ell\in L}\end{aligned}$$

where $\iota$ is a constant type; $\to$ is the functional type operator; $\tau$ $\mathsf{ref}$ is the type of locations containing a value of type $\tau$; $\{x_i{:}\tau_i\}^{i\in I}$ is a record type; and $I, J, K, L \subset \mathbb{N}$. In class types, $\{m_i : \tau_i\}$ is a record type and $[p_\ell]$ is a set of names, where $[p_\ell] \subseteq [m_i]$. In mixin types, $\{m_j : \tau_j\}, \{m_k : \tau_k\}$ are record types and $[p_\ell]$ is a set of names, where $[p_\ell] \subseteq ([m_j] \cup [m_k])$. Although record expressions and values are ordered so that we can fix an order of evaluation, record types are unordered. We also assume we have a function *typeof* from constant terms to types that respects the following *typability condition* [45]: for $const \in Const$ and value $v$, if $typeof(const) = \tau' \to \tau$ and $\emptyset \vdash v : \tau'$, then $\delta(const, v)$ is defined and $\emptyset \vdash \delta(const, v) : \tau$.

Our type system supports structural subtyping (the $<:$ relation) along with the subsumption rule ($sub$). The subtyping rules are shown in appendix B. Since subtyping on references is unsound and we wish to keep subtyping and inheritance completely separate, we have only the basic subtyping rules for function and record types. Subtyping only exists at the object level, and is not supported for class or mixin types.

Typing environments are defined as follows:

$$\Gamma ::= \varepsilon \ \mid\ \Gamma, x : \tau \ \mid\ \Gamma, \iota_1 <: \iota_2$$

$$\frac{\Gamma \vdash g : \gamma \rightarrow \{m_i : \tau_i\}^{i \in All} \rightarrow \{m_i : \tau_i\}^{i \in All}}{\Gamma \vdash \mathsf{classval}\langle g, [m_i]^{i \in All}, [p_\ell]^{\ell \in Prot}\rangle : \mathsf{class}\langle \gamma, \{m_i{:}\tau_i\}^{i \in All}, [p_\ell]^{\ell \in Prot}\rangle} \quad (\textit{class val})$$

$$\frac{\Gamma \vdash e : \mathsf{class}\langle \gamma, \{m_i{:}\tau_i\}^{i \in All}, [p_\ell]^{\ell \in Prot}\rangle}{\Gamma \vdash \mathsf{new}\ e : \gamma \rightarrow \{m_j{:}\tau_j\}^{j \in All \setminus Prot}} \quad (\textit{instantiate})$$

$$\frac{\begin{array}{lrl} (\mathsf{New}) & \text{For } j \in \textit{New}: & \Gamma \vdash v_{m_j} : \qquad \eta \rightarrow \sigma \rightarrow \tau^{\downarrow}_{m_j} \\ (\mathsf{Redef}) & \text{For } k \in \textit{Redef}: & \Gamma \vdash v_{m_k} : \tau^{\uparrow}_{m_k} \rightarrow \eta \rightarrow \sigma \rightarrow \tau^{\downarrow}_{m_k} \\ (\mathsf{Constr}) & & \Gamma \vdash c : \gamma_d \rightarrow \{\text{fieldinit} : \eta, \text{superinit} : \gamma_b\} \end{array}}{\Gamma \vdash \begin{pmatrix} \mathsf{mixin} \\ \quad \mathsf{method}\ m_j = v_{m_j}; \\ \quad \mathsf{redefine}\ m_k = v_{m_k}; \\ \quad \mathsf{protect}\ [p_\ell]; \\ \quad \mathsf{constructor}\ c; \\ \mathsf{end} \end{pmatrix}^{\substack{j \in New \\ k \in Redef \\ \ell \in Prot}} : \begin{array}{l} \mathsf{mixin}\langle \gamma_b, \gamma_d, \{m_i : \tau^{\uparrow}_{m_i}, m_k : \tau^{\uparrow}_{m_k}\}, \\ \qquad\qquad \{m_j : \tau^{\downarrow}_{m_j}, m_k : \tau^{\downarrow}_{m_k}\}, [p_\ell]\rangle \end{array}} \quad (\textit{mixin})$$

$$\text{where} \quad \begin{array}{l} \sigma = \{m_i : \tau^{\uparrow}_{m_i}, m_j : \tau^{\downarrow}_{m_j}, m_k : \tau^{\downarrow}_{m_k}\} \\ [p_\ell] \ \subseteq \ [m_i] \cup [m_j] \cup [m_k] \\ m_i, \tau^{\uparrow}_{m_i}, \tau^{\uparrow}_{m_k} \text{ are inferred from method bodies} \end{array}$$

$$\frac{\begin{array}{l} \Gamma \vdash e_1 : \mathsf{mixin}\langle \gamma_b, \gamma_d, \sigma_{\mathrm{Old}}, \sigma_{\mathrm{New}}, P_d\rangle \\ \Gamma \vdash e_2 : \mathsf{class}\langle \gamma_c, \sigma_b, P_b\rangle \\ \Gamma \vdash \sigma_d <: \sigma_b <: \sigma_{\mathrm{Old}} \\ \Gamma \vdash \gamma_b <: \gamma_c \end{array}}{\Gamma \vdash e_1 \diamond e_2 : \mathsf{class}\langle \gamma_d, \sigma_d, P_b \cup P_d\rangle} \quad (\textit{mixin app})$$

$$\text{where} \quad \begin{array}{ll} \sigma_b & = \{m_k : \tau_{m_k}, m_l : \tau_{m_l}, m_i : \tau_{m_i}\} \\ \sigma_{\mathrm{Old}} & = \{m_i : \tau^{\uparrow}_{m_i}, m_k : \tau^{\uparrow}_{m_k}\} \\ \sigma_{\mathrm{New}} & = \{m_j : \tau^{\downarrow}_{m_j}, m_k : \tau^{\downarrow}_{m_k}\} \\ \sigma_d & = \{m_i : \tau_{m_i}, m_j : \tau^{\downarrow}_{m_j}, m_k : \tau^{\downarrow}_{m_k}, m_l : \tau_{m_l}\} \end{array}$$

**Fig. 3.** Typing Rules for Class-Related Forms

where $x \in Var$, $\tau$ is a well-formed type, $\iota_1, \iota_2$ are constant types, and $x, \iota_1 \notin dom(\Gamma)$.

Typing judgments are as follows:

$$\begin{array}{ll} \Gamma \vdash \tau_1 <: \tau_2 & \tau_1 \text{ is a subtype of } \tau_2 \\ \Gamma \vdash e : \tau & e \text{ has type } \tau \end{array}$$

The set of typing rules for class-related forms is shown in Fig.3. The remaining rules are standard and can be found in appendix B.

(*class val*) rule types class values. A class value is composed of an expression and two sets of method names. The expression $g$ is the generator (see section 4) which produces a function that will later, at the time of new application, return a real object. The type of $g$ can be determined by examining the type of the class value, $\mathsf{class}\langle\gamma, \{m_i{:}\tau_i\}, [p_\ell]\rangle$. Generator $g$ takes an argument of type $\gamma$ and returns a function that will return an object once the fixed-point operator is applied. The return type of $g$ is therefore $\sigma \to \sigma$, where $\sigma$ represents the type of *self*, $\{m_i : \tau_i\}$. This record type includes *all* methods, not only public methods. When the fixed-point operator is applied, $fix(g\ v)$ will have type $\sigma$ when $v$ has type $\gamma$.

(*mixin*) rule types mixin declarations. We describe it following the order of its premises. Note that mixin methods make typing assumptions about methods of the superclass to which the mixin will be applied. We refer to these types as *expected* types since the actual superclass methods may have different types. The exact relationship between the types expected by the mixin and the actual types of the superclass methods is formalized in rule (*mixin app*). We mark types that come from the *superclass* with $^{\uparrow}$ and those that will be changed or added in the *subclass* with $^{\downarrow}$.

- (New) The bodies of the new methods $v_{m_j}$ are typed with a function type. The argument types are the type of the private field ($\eta$) and the type of *self* ($\sigma$). We do not lose generality by assuming only one field per class since $\eta$ can be a tuple or record type. The return type is $\tau^{\downarrow}_{m_j}$.
- (Redef) The bodies of the redefined methods $v_{m_k}$ are also typed with a function type. The first argument type $\tau^{\uparrow}_{m_k}$ is that of *next*, *i.e.*, the superclass method with the same name (recall that the new body can refer to the old body via *next*). The meaning of $\eta$ and $\sigma$ is the same as for the new methods. It is not known at the time of mixin definition to which class the mixin will be applied, so the actual type of the method replaced by $m_k$ may be different from the expected type $\tau^{\uparrow}_{m_k}$.
- (Constr) The constructor expression $c$ is a function that takes an argument of type $\gamma_d$ and returns a record with two components. The component labelled *fieldinit* is the initialization expression for the private field. Clearly, it has to have the same type $\eta$ as that assumed for the field when typing method bodies. The component labeled *superinit* is the expression passed as the parameter to the superclass generator. Its type $\gamma_b$ is inferred from

the constructor definition since the superclass is not available at the time of mixin definition.

Both new and redefined methods in the mixin may call superclass methods (*i.e.*, methods that are expected to be supported by any class to which the mixin is applied). We refer to these methods as $m_i$. Their types $\tau^{\uparrow}_{m_i}$ are inferred from the mixin definition.

The mixin is typed with a $\mathsf{mixin}\langle\ldots\rangle$ type, which encodes the following information about the mixin:

- $\gamma_b$ is the expected argument type of the superclass generator.
- $\gamma_d$ is the exact argument type of the mixin generator.
- $\{m_i : \tau^{\uparrow}_{m_i}, m_k : \tau^{\uparrow}_{m_k}\}$ are the expected types of the methods that must be supported by any class to which the mixin is applied. Recall that $m_i$ are the methods that are not redefined by the mixin but still expected to be supported by the superclass since they are called by other mixin methods, and $\tau^{\uparrow}_{m_k}$ are the types assumed for the old bodies of the methods redefined in the mixin.
- $\{m_j : \tau^{\downarrow}_{m_j}, m_k : \tau^{\downarrow}_{m_k}\}$ are the exact types of mixin methods (new and redefined, respectively).
- $[p_\ell]$ is an annotation listing the names of all methods to be protected, both new and redefined.

Type information contained in the $\mathsf{mixin}\langle\ldots\rangle$ type is used when typing mixin application in rule (*mixin app*).

(*mixin app*) rule types mixin-based inheritance. In the rule definition, $\sigma_b$ contains the type signatures of all methods supported by the superclass to which the mixin is applied. In particular, $m_k$ are the superclass methods redefined by the mixin, $m_i$ are the superclass methods called by the mixin methods but not redefined, and $m_l$ are the superclass methods not mentioned in the mixin definition at all. Note that the superclass may have more methods than required by the mixin constraint.

Type $\sigma_d$ contains the signatures of all methods supported by the subclass created as a result of mixin application. Methods $m_{i,l}$ are inherited directly from the superclass, methods $m_k$ are redefined by the mixin, and methods $m_j$ are the new methods added by the mixin. We are guaranteed that methods $m_j$ are not present in the superclass by the construction of $\sigma_b$ and $\sigma_d$: $\sigma_d$ is defined so that it contains all the labels of $\sigma_b$ plus labels $m_j$. Type $\sigma_{\text{Old}}$ lists the (expected) types of the superclass methods assumed when typing the mixin definition. Type $\sigma_{\text{New}}$ lists the exact types of the methods newly defined or redefined in the mixin.

The premises of the rule are as follows:

- $\mathsf{mixin}\langle\ldots\rangle$ and $\mathsf{class}\langle\ldots\rangle$ are the types of the mixin and the superclass, respectively.
- The $\sigma_d <: \sigma_b$ constraint requires that the types of the methods redefined by the mixin ($m_k$) be subtypes of the superclass methods with the same name. This ensures that all calls to the redefined methods in $m_i$ and $m_l$ (methods inherited intact from the superclass) are type-safe.

- The $\sigma_b <: \sigma_{\mathrm{Old}}$ constraint requires that the actual types of the superclass methods $m_i$ and $m_k$ be subtypes of the expected types assumed when typing the mixin definition.
- The $\gamma_b <: \gamma_c$ constraint requires that the actual argument type of the superclass generator be a supertype of the type assumed when typing the mixin definition.

In the type of the class value created as a result of mixin application, $\sigma_b$ is the argument type of the generator, and $\sigma_d$ (see above) is the type of objects that will be instantiated from the class (except for the protected methods which are included in $\sigma_d$ but hidden in the instantiated objects). In the resulting subclass we protect all methods that are protected either in the superclass or in the mixin.

The (*mixin app*) rule also determines how name clashes between the mixin and the superclass are handled. Suppose the superclass and the mixin contain a method with the same name $m$. If $m$ is a redefined method in the mixin (*i.e.*, $m \in [m_k]$), then it will replace the method from the superclass as long as its type $\tau^{\downarrow}_{m_k}$ is a subtype of the replaced method's type $\tau_{m_k}$. This is checked by the $\sigma_d <: \sigma_b$ premise. If $m$ is a new method (*i.e.*, $m \in [m_j]$), then the rule's premises will fail since a method that is considered new by the mixin appears in the superclass ($m = m_j \in \sigma_b$), and the type system will signal an error.

(*instantiate*) rule types the creation of a new object. The new $e$ term is typed as a function that takes the generator's argument and returns a fully initialized object. The object's type contains only the public methods; the protected methods are hidden.

The proof of soundness is omitted for lack of space. The complete meta-theory may be found in [8].

## 6 Related Work

In the literature, there exists an extensive body of work on calculi for object-oriented languages. Our calculus can be directly compared with the following class-oriented calculi:

- In the simplest of Cook's calculi [21], objects are represented by records of methods, and created by taking the fixed-point of the function representing the class (*constructor* in Cook's terminology). Inheritance is modeled by generating the subclass constructor from the superclass constructor, and *self* is bound early. However, classes are not a basic construct. The calculus relies on record concatenation operators, but typing issues associated with them are not addressed.
- The closure semantics version of the "dynamic inheritance" language analyzed by Kamin and Reddy [32] is similar to our calculus. The language is class-based, and the semantics of inheritance is similar to our generators. They also compare late and early *self* binding (*fixed-point model* and *self-application model* in their terminology). However, no type system is provided and there is no discussion of object construction or method encapsulation.

– The calculus of Wand [44] is class-based. Classes are modeled as extensible records, inheritance is record concatenation plus *self* update so that inherited methods refer to the correct object. As in our calculus, objects are records, *self* is bound early, and the *new* operation (called *constructor*) is an application of the fixed-point operator. In contrast to our calculus, the subclass must know and directly initialize the fields of the superclass. There is also no support for parameterized inheritance. Another solution, proposed in [40], is to rename the superclass fields, but this does not ensure consistent initialization.
– TOOPL [13] is a calculus of classes and objects. *MyType* specialization is used for inheritance, forcing late *self* binding (*i.e.*, *self* is bound each time a method is invoked, and not just once when the object is created). To ensure type safety when *MyType* appears in the method signature, there are standard constraints on method subtyping. A related work is PolyTOIL [18], where inheritance is completely separated from subtyping. Inheritance is based on *matching*, which is a relation between class interfaces that does not require method types to follow the standard constraints on recursive types, while object types employ standard subtyping. PolyTOIL also has imperative updating of object fields, but inheritance is still modeled with *MyType* in order to support binary methods. The drawback is the complexity of the type system. In [17], another language is presented, Loom, where only matching is used and the type system is simplified.

This paper is an attempt to build a simpler class-based calculus. The absence of *MyType* makes it weaker, but imperative updating appears sufficient to model the desirable features that are needed in practice.

Other approaches to modeling classes can be found in object-based calculi, where classes are not first-class expressions and have to be constructed from more primitive building blocks:

– Abadi and Cardelli have proposed encoding classes in a pure object system using records of pre-methods [1]. Pre-methods can be thought of as functions from *self* to method bodies or functions that are written as methods but not yet installed in any object. The difference between the result of *Gen* (see section 4 above) and a record of pre-methods is that the former is a function from *self* to a record of methods while the latter is a record of functions from *self* to methods. In the Abadi-Cardelli approach, a class is an object that contains a record of pre-methods and a constructor function used to package pre-methods into objects. The primary advantage of the record-of-pre-methods encoding is that it does not require a complicated form of objects. All that is needed is a way of forming an object from a list of component definitions. However, this approach provides no language support for classes, and imposes complicated constraints on the objects used as classes to obey some basic requirements for class constructs (see section 2 above and [29] for a complete account).
– Another approach to modeling classes as objects is developed by Fisher [26] in a functional setting, and by Bono and Fisher [5] in an imperative setting.

Classes are modeled as encapsulated extensible objects. Inheritance is then modeled as the method addition operation on objects, which can be in one of two states [28]: a *prototype* (can be extended but not subtyped, so prototype objects are similar to classes), and a "proper" object which is subtypable but cannot be extended. A form of a bounded existential quantifier is used to (partially) abstract the class implementation when objects are in the prototype state. While the system of [5] can model a form of mixins, our calculus is simpler, more intuitive, and has encapsulation and object creation semantics closer to those used by popular programming languages.
- Pierce and Turner [39] model classes as object-generating functions. They interpret inheritance as modification of the object-generating functions used to model classes (*existential models*). This encoding is somewhat cumbersome, since it requires programmers to explicitly manipulate `get` and `put` functions which intuitively convert the hidden state of superclass objects into that of subclass objects. Hofmann and Pierce [31] introduce a refined version of $F_{<:}$ that permits only positive subtyping. With this restriction, `get` and `put` functions are both guaranteed to exist and hence may be handled in a more automatic fashion in class encodings. In our calculus, instead of encapsulation at the object type level, we use subtyping to hide protected methods and $\lambda$-binding to hide private fields.
- The Hopkins Object Group has designed a type-safe class-based object-oriented language with a rich feature set called I-LOOP [23]. Their type system is based on polymorphic recursively constrained types, for which they have a sound type inferencing algorithm. The main advantage of this approach is the extreme flexibility afforded by recursively constrained types. However, inferred types are large and difficult to read.

Bruce et al. [16] show how the main approaches to modeling objects can be seen in a unified framework. The state of the art in modeling classes is not as well established. We hope that this work might be a step in this direction.

To the best of our knowledge, there are not many formal settings in which mixin-based inheritance is analyzed.

- Flatt et al. implement mixins in the MzSCHEME language [25] and formalize an extension of a subset of JAVA with mixins in [30]. Their system supports higher-order mixin composition, a hierarchy of named interface types, and resolution of accidental name collisions. The collision resolution system allows old and new method definitions to coexist. The two are distinguished using the "view" of an object, which is carried with the object at run-time and altered at each subsumption step. As a result, method lookup is sensitive to the object's history of subsumptions. In contrast to the system of [30], our calculus is not based on any particular language. Our mixins are created and manipulated as run-time values as opposed to static top-level declarations. Mixin constraints prevent objects from having incompatible methods with the same name, so method lookup is straightforward and does not depend on the object's subsumption history. Proper object initialization is guaranteed.

– BETA [36] replaces classes, procedures, functions, and types by a single abstraction mechanism called the *pattern*. Objects are created from the patterns, and in addition to traditional objects as found in conventional object-oriented languages, objects in BETA may also represent function activations, exception occurrences, or concurrent processes. Patterns may be used as *superpatterns* to other patterns in a manner similar to conventional inheritance. Since patterns are a general concept, inheritance is available also for procedures, functions, exceptions, coroutines, and processes. *Virtual* patterns are similar to generic templates or parameterized classes with the additional benefit that the parameter may be restricted without actually instantiating the template (this is similar to computing the mixin constraint without actually applying the mixin to a class). Mixin inheritance is a partial case of the very general pattern inheritance mechanism developed in BETA.
– OCAML [34] supports a very limited form of parameterized inheritance by combining a module abstraction mechanism with classes that can inherit across module boundaries. Because the exact module containing the superclass may not be known when the subclass is defined, the same subclass can be used with multiple superclass definitions. However, methods not mentioned in the superclass type become inaccessible. In the example of section 2.4, this would mean that all methods that are present in the Socket $\diamond$ *Object* class besides *read* and *write* are forgotten once Encrypted mixin is applied to it.
– Ancona and Zucca [2] study a rigorous semantics foundations for mixins independently from the notions of classes and objects, starting from an algebraic setting for module composition. It may be possible to apply their techniques to the study of the algebraic semantics of our calculus.

## 7 Conclusions and Future Work

The main strengths of our calculus are its simplicity and its power in modeling mixin inheritance. Both the operational semantics and the type system are structured to combine new rules for *class*-based features (mixins, classes, and instantiation) with standard rules for *object*-based features (represented by records, functions, and assignable locations). We also preserve such properties as encapsulation (private fields, protected methods) and modularity (minimized dependencies of a subclass on superclasses, modular object creation, automatic propagation of changes in the superclass to all subclasses). All of these are desirable features for a formalism used to model classes [29]. Our mixin construct provides a formal model for a flexible inheritance mechanism, capable of expressing single inheritance, most uses of multiple inheritance, and also new uses of inheritance such as applying the same mixin more than once.

Some of the design choices may appear debatable, *e.g.*, the decision not to support *super* in the calculus. While a redefined method can refer to the old method body via *next*, other methods have no way of calling it. This decision was motivated mainly by our desire to support an efficient implementation, and, in fact, the calculus can be easily extended to support *super* by keeping a reference to the entire superclass object (*supergen self*) instead of selecting the component being redefined (see section 4). Also debatable is the decision to support imperative instead of functional object updates. This choice was motivated by our desire for simplicity and the relative complexity of supporting functional update (*e.g.*, the need for *MyType*).

We believe that our calculus can be considered a step towards a better understanding of class-based languages, both because it shows how support for modular programming techniques can be included in a sound calculus without compromising its simplicity, and because it can serve as a starting point for more foundational studies such as denotational semantics for the class and mixin constructs. Topics for future research include developing an efficient implementation of the core calculus and extending it to a full language; studying an extension of the core calculus with ML polymorphism in order to combine classes and objects with the full power of ML type inference; combining existential types with our simple object types to provide a form of implementation types; and expanding our rules for mixins to account for higher-order mixins.

## References


[1] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, 1996.

[2] D. Ancona and E. Zucca. An algebraic approach to mixins and modularity. In *Proc. Algebraic and Logic Programming (ALP)*, pages 179–193. LNCS 1139, Springer-Verlag, 1996.

[3] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[4] V. Bono and M. Bugliesi. Matching for the lambda calculus of objects. *Theoretical Computer Science*, 1998. To appear.

[5] V. Bono and K. Fisher. An imperative, first-order calculus with object extension. In *Proc. ECOOP '98*, pages 462–497. LNCS 1445, Springer-Verlag, 1998. Preliminary version appeared in FOOL 5 proceedings.

[6] V. Bono and L. Liquori. A subtyping for the Fisher-Honsell-Mitchell lambda calculus of objects. In *Proc. CSL '94*, pages 16–30. LNCS 933, Springer-Verlag, 1995.

[7] V. Bono, A. Patel, V. Shmatikov, and J. C. Mitchell. A core calculus of classes and objects. In *Proc. 15th Conference on the Mathematical Foundations of Programming Semantics (MFPS '99)*, 1999. To appear.

[8] V. Bono, A. Patel, V. Shmatikov, and J. C. Mitchell. A core calculus of object, classes, and mixins. Technical Report, The University of Birmingham and Stanford University, 1999. Forthcoming.

[9] N. Boyen, C. Lucas, and P. Steyaert. Generalized mixin-based inheritance to support multiple inheritance. Technical Report vub-prog-tr-94-12, Vrije Universiteit Brussel, 1994.

[10] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, University of Utah, 1992.

[11] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA '90*, pages 303–311, 1990.

[12] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proc. International Conference on Computer Languages (ICCL '92)*, pages 282–290, 1992.

[13] K. B. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Proc. POPL '93*, pages 285–298, 1993.

[14] K. B. Bruce. A paradigmatic object-oriented language: design, static typing and semantics. *J. Functional Programming*, 4(2):127–206, 1994.

[15] K. B. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):217–238, 1995.

[16] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. In *Proc. TACS '97*, pages 415–438. LNCS 1281, Springer-Verlag, 1997.

[17] K. B. Bruce, L. Petersen, and A. Finch. Subtyping is not a good "match" for object-oriented languages. In *Proc. ECOOP '97*, pages 104–127. LNCS 1241, Springer-Verlag, 1997.

[18] K. B. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proc. ECOOP '95*, pages 26–51. LNCS 952, Springer-Verlag, 1995.

[19] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[20] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proc. POPL '90*, pages 125–135, 1990.

[21] W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[22] E. Crank and M. Felleisen. Parameter-passing and the lambda calculus. In *Proc. POPL '91*, pages 233–244, 1991.

[23] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proc. OOPSLA '95*, pages 169–184, 1995.

[24] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[25] R. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. ICFP '98*, pages 94–104, 1998.

[26] K. Fisher. *Type Systems for Object-Oriented Programming Languages*. PhD thesis, Stanford University, 1996.

[27] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda-calculus of objects and method specialization. *Nordic J. of Computing*, 1(1):3–37, 1994. Preliminary version appeared in Proc. *LICS '93*, pp. 26–38.

[28] K. Fisher and J. C. Mitchell. A delegation-based object calculus with subtyping. In *Proc. 10th International Conference on Fundamentals of Computation Theory (FCT '95)*, pages 42–61. LNCS 965, Springer-Verlag, 1995.

[29] K. Fisher and J. C. Mitchell. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems*, 4(1):3–26, 1998. Preliminary version appeared in Marktoberdorf '97 proceedings.

[30] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. POPL '98*, pages 171–183, 1998.

[31] M. Hofmann and B. C. Pierce. Positive subtyping. *Information and Computation*, 126(1):11–33, 1996. Preliminary version appeared in Proc. *POPL '95*.

[32] S. Kamin and U. Reddy. Two semantic models of object-oriented languages. In C. Gunther and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994.

[33] S. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, 1989.

[34] X. Leroy, D. Rémy, J. Vouillon, and D. Doligez. The Objective Caml system, documentation and user's guide. `http://caml.inria.fr/ocaml/htmlman/`, 1999.
[35] M. Van Limberghen and T. Mens. Encapsulation and composition as orthogonal operators on mixins: a solution to multiple iinheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.
[36] O. Lehrmann Madsen, B. Moller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Language*. Addison-Wesley, 1993.
[37] I. Mason and C. Talcott. Programming, transforming, and proving with function abstractions and memories. In *Proc. ICALP '89*, pages 574–588. LNCS 372, Springer-Verlag, 1989.
[38] D. Moon. Object-oriented programming with Flavors. In *Proc. OOPSLA '86*, pages 1–8, 1986.
[39] B. C. Pierce and D. N. Turner. Simple type-theoretic foundations for object-oriented programming. *J. Functional Programming*, 4(2):207–248, 1994. Preliminary version appeared in Proc. *POPL '93* under the title *Object-Oriented Programming Without Recursive Types*.
[40] U. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. Conference on Lisp and Functional Programming*, pages 289–297, 1988.
[41] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. ECOOP '98*, pages 550–570, 1998.
[42] B. Stroustrup. *The C++ Programming Language (3rd ed.)*. Addison-Wesley, 1997.
[43] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proc. OOPSLA '96*, pages 359–369, 1996.
[44] M. Wand. Type inference for objects with instance variables and inheritance. In C. Gunther and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994.
[45] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

# A Definition of Contexts

The definition of contexts is standard but lengthy due to the number of subexpressions in the mixin expression:

$$
\begin{aligned}
C ::= {} & [\,] \mid C\ e \mid e\ C \mid \lambda x.C \mid C.x \mid C \diamond e \mid e \diamond C \\
& \mid \{m_1 = e_1, \ldots, m_{i-1} = e_{i-1}, m_i = C, m_{i+1} = e_{i+1}, \ldots, m_n = e_n\}^{1 \le i \le n} \\
& \mid \mathsf{H}\ h.C \mid \mathsf{H}\langle x, C\rangle h.e \mid \mathsf{new}\ C \mid \mathsf{classval}\langle C, \mathcal{M}, \mathcal{P}\rangle
\end{aligned}
$$

$$
\begin{array}{lll}
\mid \mathsf{mixin}^{\substack{j \in New \\ k \in Redef \\ \ell \in Prot}} & \mid \mathsf{mixin}^{\substack{j \in New \setminus [i] \\ k \in Redef \\ \ell \in Prot}} & \mid \mathsf{mixin}^{\substack{j \in New \\ k \in Redef \setminus [i] \\ \ell \in Prot}} \\
\quad \mathsf{method}\ m_j = v_{m_j}; & \quad \mathsf{method}\ m_j = v_{m_j}; & \quad \mathsf{method}\ m_j = v_{m_j}; \\
\quad \mathsf{redefine}\ m_k = v_{m_k}; & \quad \mathsf{method}\ m_i = C; & \quad \mathsf{redefine}\ m_k = v_{m_k}; \\
\quad \mathsf{protect}\ [p_\ell]; & \quad \mathsf{redefine}\ m_k = v_{m_k}; & \quad \mathsf{redefine}\ m_i = C; \\
\quad \mathsf{constructor}\ C; & \quad \mathsf{protect}\ [p_\ell]; & \quad \mathsf{protect}\ [p_\ell]; \\
\mathsf{end} & \quad \mathsf{constructor}\ v_c; & \quad \mathsf{constructor}\ v_c; \\
 & \mathsf{end} & \mathsf{end}
\end{array}
$$

## B Type Rules

The type rules for class-related forms were presented in section 5. The remaining type rules are presented here.

### B.1 Subtyping Rules

The subtyping rules are standard. Objects support both depth and width subtyping.

$$\frac{}{\Gamma, \iota_1 <: \iota_2 \vdash \iota_1 <: \iota_2} \quad (<: \; proj) \qquad \frac{}{\Gamma \vdash \tau <: \tau} \quad (refl)$$

$$\frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3} \quad (trans) \qquad \frac{\Gamma \vdash \tau' <: \tau \quad \Gamma \vdash \sigma <: \sigma'}{\Gamma \vdash \tau \rightarrow \sigma <: \tau' \rightarrow \sigma'} \quad (arrow)$$

$$\frac{\Gamma \vdash \tau_i <: \sigma_i \quad i \in I \quad I \subseteq J}{\Gamma \vdash \{m_i : \tau_i\}^{i \in I} <: \{m_j : \sigma_j\}^{j \in J}} \quad (<: \; record)$$

### B.2 Type Rules for Expressions

The type rules for expressions other than class-related forms are simple, except for heaps, which have to be typed globally.

$$\frac{typeof(const) = \tau}{\Gamma \vdash const : \tau} \quad (const) \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau} \quad (proj) \qquad \frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x.e : \tau \rightarrow \sigma} \quad (\lambda)$$

$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \; e_2 : \sigma} \quad (app) \qquad \frac{}{\Gamma \vdash fix : (\sigma \rightarrow \sigma) \rightarrow \sigma} \quad (fix)$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau <: \sigma}{\Gamma \vdash e : \sigma} \quad (sub) \qquad \frac{\Gamma \vdash e_i : \tau_i}{\Gamma \vdash \{x_i = e_i\}^{i \in I} : \{x_i : \tau_i\}} \quad (record)$$

$$\frac{\Gamma \vdash e : \{x : \sigma\}}{\Gamma \vdash e.x : \sigma} \quad (lookup) \qquad \frac{}{\Gamma \vdash \mathsf{ref} : \tau \rightarrow \tau \; \mathsf{ref}} \quad (ref) \qquad \frac{}{\Gamma \vdash \; ! : \tau \; \mathsf{ref} \rightarrow \tau} \quad (!)$$

$$\frac{}{\Gamma \vdash \; := \;\; : \tau \; \mathsf{ref} \rightarrow \tau \rightarrow \tau} \quad (:=)$$

$$\frac{\Gamma' = \Gamma, x_1 : \tau_1 \; \mathsf{ref}, \ldots, x_n : \tau_n \; \mathsf{ref} \quad \Gamma' \vdash v_i : \tau_i \quad \Gamma' \vdash e : \tau}{\Gamma \vdash H\langle x_1, v_1 \rangle \ldots \langle x_n, v_n \rangle . e : \tau} \quad (heap)$$