Electronic Notes in Theoretical Computer Science 20 (1999)
URL: `http://www.elsevier.nl/locate/entcs/volume20.html` 22 pages

A Core Calculus of Classes and Objects

Viviana Bono [a], Amit Patel [b], Vitaly Shmatikov [b], and John Mitchell [b]

[a] *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`*)*

[b] *Computer Science Department, Stanford University, Stanford, CA 94305-9045, U.S.A.,* {`amitp,shmat,jcm`}`@cs.stanford.edu`

Abstract

We present an imperative calculus for a class-based language. By introducing classes as the basic object-oriented construct in a λ-calculus with records and references, we obtain a system with an intuitive operational semantics. Objects are instantiated from classes and represented by records. The type system for objects uses only functional, record, and reference types, and there is a clean separation between subtyping and inheritance. We demonstrate that the calculus is sound and sufficiently expressive to model advanced language features such as inheritance with method redefinition, multi-level encapsulation, and modular object construction.

Key words: object-oriented language, class, calculus, operational semantics, type system.

1 Introduction

While popular object-oriented programming languages such as C++ [29] and Java [2] are overwhelmingly class-based, most previous core calculi for object-oriented languages were based on objects. In our framework, classes are the basic construct. 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 [15,9,6,22]. This makes it difficult to develop simple typed calculi of objects that include object subtyping and allow natural forms of class-based programming techniques.

The calculus presented in this paper can be seen both as a proposal for a language design and as a step towards a foundational study of class-based languages. It supports 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. As a consequence, the type system involves only functional types, record types, and reference types. In particular, we do not need polymorphic object types or recursive *MyType*.

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 (sections 5 and 6). In section 7, we compare our calculus with other object-oriented calculi.

2 Design of the Core Calculus

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

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 requires 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, structural subtyping, and modular object creation.

Data encapsulation. Typically, a class in an object-oriented program may have two uses: *inheritance* (creating a new subclass as an extension of the existing class) and *instantiation* (creating a new object from the class). Therefore, a class has two kinds of users: *subclass implementors* and *object users*, *i.e.*, users of objects instantiated from the class. It is necessary to distinguish between the methods and fields that are available only within the implementation of the class itself, those available only to subclasses, and those available to both subclasses *and* object users.

Any information that is available only to the class implementation we call private; information available to both the class and its subclasses we call protected; and information available to the class, subclasses, and the object users we call public. Note that unlike C++ and some approaches to encapsulation in object calculi (*e.g.*, existential types), our private means that the information is invisible, not merely inaccessible. We believe that this is a better approach, since *no* information about data representation is revealed — not even the number and names of fields. In contrast, C++ signals different errors for trying to access a field that does not exist and a field that is private in one of the superclasses (the latter field is visible, but not accessible), so the number and names of fields can be determined.

Structural subtyping. As in most popular object-oriented languages, objects in our calculus can only be created by instantiating a class. We use *structural subtyping* to remove the dependency of object users on class implementation. Each object has an *object type*, which lists the names and types of methods and fields but does *not* include information about the class from which the object was instantiated. Therefore, objects created from unrelated classes can be substituted for each other if their types satisfy the subtyping relation.

Structural subtyping was a deliberate design decision motivated by our desire to minimize code dependencies between object users and class implementors. A different approach would be to follow C++, in which an object's type is related to the class from which it was instantiated, and subtyping relations apply only to objects instantiated from the same class hierarchy.

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 must 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. Each class implementor writes the local constructor for his own class. The operational semantics rules of our calculus reduce class expressions to generator functions, which call all constructors in the inheritance chain in the correct order, producing a function that returns a correctly 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 [22].

"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 [14], in which inheritance was modeled by record subtyping. Unlike records, however, object methods should be able to modify fields and invoke sibling methods [16]. 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. *MyType* can be supported by using row variables combined with recursive types [21,20,22], match-bound type variables [13,4], 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 [22,13].

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 [10]. 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 Overview of the Core Calculus

The 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 λ-binding are standard, we need not introduce new operational semantics or type rules for objects. Instead, we introduce new constructs and rules only for classes. The new constructs are: *class values* (representing complete classes), *class extension 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 public method names, and the set of protected method names. 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.

Class extension expressions (extend *expr* with...) model inheritance by defining one class (subclass) as an extension of another (superclass). When evaluated, extend produces a generator for the subclass which takes the record of superclass methods built by the superclass generator and modifies it by adding and/or replacing methods.

The instantiation expression (new), when applied to a class value and evaluated, produces a new object from the class value by invoking the class generator and applying the fixed-point operator to its result.

Objects being records in our calculus, object types are treated in the same way as record types, with subtyping relations inferred from object types rather than declared explicitly by the programmer or extracted from the class hierarchy. Objects from different class hierarchies can be used interchangeably because the type of an object retains no connection to the class from which it was instantiated.

For simplicity, the core calculus includes only private fields and public and protected methods. Private methods can be modeled by using 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. For simplicity, the core calculus has exactly one private field per class, which may have a record type. Each method body takes the class's private field as a parameter which is bound when the class generator constructs a new object.

2.4 An Example

The following example demonstrates how a class definition can be written in our calculus.

let EncryptedFile = extend File with
  protected decrypt = λkey. λself. λdata. $\cdots$;
  protected encrypt = λkey. λself. λdata. $\cdots$;

method read = λnext. λkey. λself. λnbytes. self.decrypt(next(nbytes));
method write = λnext. λkey. λself. λdata. next(self.encrypt(data));
constructor λ(key, filename).{ fieldinit=key, superinit=filename };
end in $\cdots$

Note that in our calculus, all user-defined classes must be defined as an extension of another class (with a predefined empty class at the root of the hierarchy). No special namespaces or declaration forms other than class extension expressions are required.

Each class extension expression contains public methods (marked by the method keyword), protected methods (protected keyword), and constructors. Instead of field declarations, every class contains a single private field which is λ-bound in each method body (λkey.$\cdots$). While a full language would support field declarations, we keep the calculus simple by not introducing a special construct for field declarations.

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

To make the calculus as modular as possible, all syntactic constructs are local: method definitions, constructors, private/protected distinctions refer only to the class itself and not to other parts of the class hierarchy. Reduction rules from Section 4 combine local information from all classes in the inheritance chain into a single generator function which is used to instantiate new objects.

3 Syntax of the Core Calculus

The syntax of our calculus is fundamentally class-based. There are three expressions involving classes: class, extend, 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; instead, these expression forms are used only to define the semantics of programs. Class values are created by class extension, 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

[1] Introducing ref, !, := as operators rather than standard forms such as ref$e$, !e, :=$e_1 e_2$, simplifies the definition of evaluation contexts and proofs of properties. As noted in [31],

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); $[m_i], [m_j]$ are sets of identifiers; and $I, \mathit{Pub}, \mathit{Prot} \subset \mathbb{N}$.

Our calculus takes a standard calculus of functions, records, and imperative features and adds new constructs to support classes. We chose to extend *Reference ML* [31], 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.

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 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:

- $\mathsf{class}\langle v_g, [m_i]^{i\in \mathit{Pub}}, [m_j]^{j \in \mathit{Prot}}\rangle$ is a *class value*, *i.e.*, the result of evaluating a class extension expression extend. 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 public methods defined in the class, and the $[m_j]$ set contains the names of protected methods.
- extend e_s with
  method $m_i = v_{m_i}$; $^{(i \in \mathit{Pub})}$
  protected $m_j = v_{m_j}$; $^{(j \in \mathit{Prot})}$
  constructor v_c;
  end
  is a *class extension*, in which e_s represents the superclass, $m_i = v_{m_i}$ are public method declarations, and $m_j = v_{m_j}$ are protected method declarations. Each method body expression $v_{m_{i,j}}$ is a function of *self*, which will be bound to the newly created object at instantiation time, and of the private *field*. If the method redefines a superclass method with the same name, then $v_{m_{i,j}}$ is also a function of *next*, which will be bound to the old method of the superclass. The v_c value in the constructor clause is a function that returns a record of two components. When evaluating an extend expression, v_c is used to build the generator as described in section 4.
- 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:

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

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

---

this is just a syntactic convenience, as is the curried version of $:=$.

Expressions:

$$
\begin{array}{rl}
e ::= & \mathit{const} \mid x \mid \lambda x.e \mid e_1\ e_2 \mid \mathit{fix}\ \mid \mathsf{ref} \mid\ ! \mid := \\
& \mid \{x_i = e_i\}^{i \in I} \mid e.x \mid \mathsf{H}\ h.e \\
& \mid \mathsf{new}\ e \mid \mathsf{class}\langle v_g, [m_i]^{i \in Pub}, [m_j]^{j \in Prot}\rangle \\
& \mid\ \mathsf{extend}\ e_s\ \mathsf{with} \\
& \qquad \mathsf{method}\ m_i = v_{m_i};\quad {}^{(i \in Pub)} \\
& \qquad \mathsf{protected}\ m_j = v_{m_j};\quad {}^{(j \in Prot)} \\
& \qquad \mathsf{constructor}\ v_c; \\
& \quad \mathsf{end}
\end{array}
$$

Values:

$$
\begin{array}{rl}
v ::= & \mathit{const} \mid x \mid \lambda x.e \mid \mathit{fix} \mid \mathsf{ref} \mid\ ! \mid := \mid :=v \mid \{x_i = v_i\}^{i \in I} \\
& \mid \mathsf{class}\langle v_g, [m_i]^{i \in Pub}, [m_j]^{j \in Prot}\rangle
\end{array}
$$

Fig. 1. Syntax of the core calculus

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

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. All classes that do not have a user-declared superclass are considered to inherit from Object. Therefore, we can simplify the calculus by assuming that every user-defined class has a superclass, *i.e.*, every class is built using an extend expression.

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 α-conversion to binders λ and H.

4 Operational Semantics

The operational semantics for our calculus extends that of *Reference ML* [31]. Reduction rules are given in Fig.2, where R are *reduction contexts* [17,19,26]. Expression *Gen* is defined below. Relation $\twoheadrightarrow$ is the reflexive, transitive, contextual closure of $\rightarrow$ 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

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

$$
\left(\begin{array}{l}
\mathsf{extend\ class}\langle g, \mathcal{M}, \mathcal{P}\rangle\ \mathsf{with}\\
\ \ \mathsf{method}\ m_i = v_{m_i};\quad ^{(i \in \mathit{Pub})}\\
\ \ \mathsf{protected}\ m_j = v_{m_j};\quad ^{(j \in \mathit{Prot})}\\
\ \ \mathsf{constructor}\ c;\\
\mathsf{end}
\end{array}\right) \to \mathsf{class}\langle \mathit{Gen}, [m_i] \cup \mathcal{M}, [m_j] \cup \mathcal{P}\rangle \quad (\text{class})
$$

(if $[m_i] \cap [m_j] = \emptyset$, $[m_j] \cap \mathcal{M} = \emptyset$, and $[m_i] \cap \mathcal{P} = \emptyset$; Gen is defined below)

Fig. 2. Reduction Rules

are defined as follows:

$$
\begin{array}{rl}
R ::= & [\ ] \mid R\ e \mid v\ R \mid R.x \mid \mathsf{new}\ R\\
\mid & \{m_1 = v_1, \ldots, m_{i-1} = v_{i-1}, m_i = R, m_{i+1} = e_{i+1}, \ldots, m_n = e_n\}^{1 \le i \le n}\\
\mid & \mathsf{extend}\ R\ \mathsf{with}\\
& \ \ \mathsf{method}\ m_i = v_{m_i};\\
& \ \ \mathsf{protected}\ m_j = v_{m_j};\\
& \ \ \mathsf{constructor}\ c;\\
& \mathsf{end}
\end{array}
$$

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 δ-typability condition.

Rules (β_v) and (*select*) are standard.

Rules (*ref*), (*deref*) and (*assign*) evaluate imperative expressions following the linear order given by the reduction context R and acting on the heap. They are formulated after [31]: (*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.

Rules (*lift*) and (*merge*) 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.

Rule (*class*) evaluates class extension expressions. An extend expression can be seen intuitively as a class declaration. The $\mathsf{class}\langle g, \mathcal{M}, \mathcal{P}\rangle$ triple represents the superclass, where g is the superclass generator, and $\mathcal{M}$ and $\mathcal{P}$ are, respectively, the names of the public and protected superclass methods. The extend expression is reduced to a triple $\langle \mathit{Gen}, [m_i] \cup \mathcal{M}, [m_j] \cup \mathcal{P}\rangle$ where Gen

is the subclass generator function defined below, $[m_i] \cup \mathcal{M}$ is the set of public method names, and $[m_j] \cup \mathcal{P}$ is the set of protected method names. Using generators delays full inheritance resolution until object instantiation time when *self* becomes available. We define $\mathcal{A}_s = \mathcal{M} \cup \mathcal{P}$ to be the names of methods inherited from the superclass (*superclass methods*), and $\mathcal{A}_e = [m_i] \cup [m_j]$ to be the names of methods defined in the current extend expression (*subclass methods*).

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).

$$
\begin{array}{l}
\mathit{Gen} \stackrel{\triangle}{=} \lambda x. \\
\qquad\qquad \mathsf{let}\ t = c(x) \quad \mathsf{in} \\
\qquad\qquad \mathsf{let}\ \mathit{supergen} = g(t.\mathrm{superinit}) \quad \mathsf{in} \\
\qquad\qquad \lambda \mathit{self}. \\
\qquad\qquad\quad \left\{ \begin{array}{lllr} m_l = \lambda y. & (\mathit{supergen\ self}).m_l & & y \\ m_k = \lambda y. v_{m_k} & & t.\mathrm{fieldinit}\ \mathit{self} & y \\ m_r = \lambda y. v_{m_r} & (\mathit{supergen\ self}).m_r & t.\mathrm{fieldinit}\ \mathit{self} & y \end{array} \right\}_{\substack{m_l \in \mathcal{A}_s \setminus \mathcal{A}_e, \\ m_k \in \mathcal{A}_e \setminus \mathcal{A}_s, \\ m_r \in \mathcal{A}_e \cap \mathcal{A}_s}}
\end{array}
$$

In the extend 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 subclass and the superclass. *Gen* uses sets of method names carried in the class value to distinguish which methods are inherited intact from the superclass and which are redefined by the subclass. To understand how the record of methods is created, recall that method bodies take parameters *field*, *self*, and, if it's a redefinition, *next*. Methods $m_l \in \mathcal{A}_s \setminus \mathcal{A}_e$ are the inherited superclass methods: they are taken intact from the superclass's object (*supergen self*). Methods $m_k \in \mathcal{A}_e \setminus \mathcal{A}_s$ are the new subclass methods: they appear for the first time in the current extend expression. *Gen* must bind *field* and *self* for them. Methods $m_r \in \mathcal{A}_s \cap \mathcal{A}_e$ are contained both in the superclass and the subclass. The subclass method overrides the superclass method (*Gen* uses the new method body v_{m_r} from the subclass definition), but it can still refer to the old method via the *next* parameter, which is bound to (*supergen self*)$.m_r$ by *Gen*. They also receive a binding for *field* and *self*. For all three sorts of methods, the method bodies are wrapped inside $\lambda y. \cdots y$ (η-expansion) to delay evaluation in our call-by-value calculus.

$$\frac{\Gamma \vdash g : \gamma \to \{m_k : \tau_k\}^{k\in Pub\cup Prot} \to \{m_k : \tau_k\}^{k\in Pub\cup Prot}}{\Gamma \vdash \mathsf{class}\langle g, [m_i]^{i\in Pub}, [m_j]^{j\in Prot}\rangle : \mathsf{class}\langle \gamma, \{m_i{:}\tau_i\}^{i\in Pub}, \{m_j{:}\tau_j\}^{j\in Prot}\rangle} \quad (\mathit{class\ val})$$

$$\frac{\Gamma \vdash e : \mathsf{class}\langle \gamma, \{m_i{:}\tau_i\}^{i\in Pub}, \{m_j{:}\tau_j\}^{j\in Prot}\rangle}{\Gamma \vdash \mathsf{new}\ e : \gamma \to \{m_i{:}\tau_i\}^{i\in Pub}} \quad (\mathit{instantiate})$$

$$\frac{\begin{array}{lrl} (\mathsf{Super}) & \forall i \in Pub_s, j \in Prot_s\colon & \Gamma \vdash e_s : \mathsf{class}\langle \rho, \{m_i{:}\tau_{m_i}\}, \{m_j{:}\tau_{m_j}\}\rangle \\ (\mathsf{New}) & \forall i \in All_e \setminus All_s\colon & \Gamma \vdash v_{m_i} : \quad\quad \eta \to \sigma \to \sigma_{m_i} \\ (\mathsf{Redef}) & \forall i \in All_e \cap All_s\colon & \Gamma \vdash v_{m_i} : \tau_{m_i} \to \eta \to \sigma \to \sigma_{m_i} \\ & & \Gamma \vdash \sigma_{m_i} <: \tau_{m_i} \\ (\mathsf{Constr}) & & \Gamma \vdash c : \gamma \to \{\text{fieldinit} : \eta, \text{superinit} : \rho\} \end{array}}{\Gamma \vdash \left(\begin{array}{l} \mathsf{extend}\ e_s\ \mathsf{with} \\ \ \mathsf{method}\ m_i = v_{m_i}; \\ \ \mathsf{protected}\ m_j = v_{m_j}; \\ \ \mathsf{constructor}\ c \\ \mathsf{end} \end{array}\right)^{i\in Pub_e, j\in Prot_e} : \mathsf{class}\langle \gamma, \{m_i{:}\sigma_{m_i}\}^{i\in Pub}, \{m_j{:}\sigma_{m_j}\}^{j\in Prot}\rangle} \quad (\mathit{extend})$$

$$\text{where} \quad \begin{array}{ll} All_s = Pub_s \cup Prot_s & Pub \ = Pub_s \cup Pub_e \\ All_e = Pub_e \cup Prot_e & Prot = Prot_s \cup Prot_e \end{array}$$

Fig. 3. Typing Rules for Class-Related Forms

Rule (*fix*) is standard.

Rule (*new*) builds a function that can create a new object. The resulting function can be thought of as the composition of three functions: $Sub \circ \mathit{fix} \circ g$. 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 [16]) is applied to bind *self* in method bodies and create a recursive record. Finally, we apply $Sub_{\mathcal{M}\cup\mathcal{P}\to\mathcal{M}}$, 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 to exclusively to class-related constructs (class, extend and new). The only operation on objects is method selection which is typed as ordinary record component selection.

Types are as follows:
$\tau ::= \iota \mid \tau_1 \to \tau_2 \mid \{x_i : \tau_i\}^{i\in I} \mid \mathsf{class}\langle \tau, \{m_i{:}\tau_i\}^{i\in Pub}, \{m_j{:}\tau_j\}^{j\in Prot}\rangle \mid \tau\ \mathsf{ref}$
where ι is a constant type; $\to$ is the functional type operator; $\{x_i : \tau_i\}^{i\in I}$ is

a record type; $\{m_i : \tau_i\}^{i \in Pub}$ and $\{m_j : \tau_j\}^{j \in Prot}$ are record types with disjoint labels ($Pub \cap Prot = \emptyset$); $I, Pub, Prot \subset \mathbb{N}$; and τ ref is the type of locations containing a value of type τ. 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* [31]: 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 ($<:$ 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 (*i.e.*, record) level, and is not supported for class types.

Typing environments are defined as follows:

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

where $x \in Var$, τ is a well-formed type, ι_1, ι_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.

In the (*class val*) rule, one can observe the basic pattern of a class value and its type. 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, $\mathsf{class}\langle\gamma, \{m_i{:}\tau_i\}^{i \in Pub}, \{m_j{:}\tau_j\}^{j \in Prot}\rangle$. Generator g takes an argument of type γ 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 σ represents the type of *self*, $\{m_k : \tau_k\}^{k \in Pub \cup Prot}$. This record type includes *all* methods, not only public methods. When the fixed-point operator is applied, $fix(g\ v)$ will have type σ when v has type γ.

Rule (*extend*) is relatively long but straightforward. We describe it following the order of its premises:

- (Super) The superclass expression e_s is expected to be typed as a class, where ρ is the argument type of the superclass generator, and the two sets of names are, respectively, the names of the public and protected superclass methods.

- (New) The bodies of the new methods are typed with a function type. The argument types are the type of the private field (η) and the type of *self* (σ). We do not lose generality by assuming only one field per class since η can be a tuple or record type.
- (Redef) The bodies of the redefined methods are also typed with a function type. The first argument type τ_{m_i} 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 η and σ is the same as for the new methods. Redefined method bodies may have a more specific type than the old ones (premise $\Gamma \vdash \sigma_{m_i} <: \tau_{m_i}$).
- (Constr) The constructor expression c is a function that takes an argument of type γ and returns a record with two components. The component labeled *fieldinit* is the initialization expression for the private field. Clearly, it has to have the same type η 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. It has to have the same type ρ as the argument of the superclass generator. Recall that because of encapsulation, a class in our calculus is unable to directly initialize the field it inherits from its superclass and has to call the superclass generator to do so.

When a class is extended, the parameter type for the subclass generator may be completely different from that of the superclass generator. However, the set of method names in the subclass must be a superset of the set of method names in the superclass. In other words, the subclass is unable to remove methods inherited from the superclass.

The $\mathsf{class}\langle\ldots\rangle$ type given to the extend expression encodes all information about the newly created class. It contains the type γ of the parameter to the generator, the names and types of public methods, and the names and types of protected methods.

Rule (*instantiate*) 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.

6 The Core Calculus is Sound

We prove that our calculus is sound via a subject reduction lemma.

Lemma 6.1 (Subject Reduction) *If $\Gamma \vdash e : \tau$ is derivable and $e \twoheadrightarrow e'$, then $\Gamma \vdash e' : \tau$ is also derivable.* □

We then introduce the notion of *faulty programs* which are the programs that reach a "stuck state" during the evaluation process, and prove that if a given program p does not diverge, then either it returns an answer, or p reduces to a faulty program. By using the subject reduction property and proving that faulty programs are not typable, we show that if a program has a type in our system, then it evaluates to an answer, under the condition that the program does not diverge.

Definition 6.2 [Faulty Programs] A program is *faulty* if it contains a subexpression of any of the following forms:

$$\begin{array}{rll}
\mathit{const}\ v & \text{where} & \mathit{const} \in \mathit{Const},\ v \in \mathit{Val},\ \delta(\mathit{const}, v) \text{ undefined} \\
v_1 v_2 & \text{where} & v_1, v_2 \text{ are values and } v_1 \neq \lambda x.e, \mathsf{ref}, !, :=, :=v \\
!v & \text{where} & v \text{ is a value and } v \neq x \\
:=v & \text{where} & v \text{ is a value and } v \neq x \\
\mathsf{H}h\langle x, v_2\rangle.C[xv_1] & \text{where} & v_1, v_2 \text{ are values} \\
\mathsf{new}\ v & \text{where} & v \text{ is a value and } v \neq \mathsf{class}\langle_,_,_\rangle \\
\mathsf{extend}\ v\ \mathsf{with} \ldots \mathsf{end} & \text{where} & v \text{ is a value and } v \neq \mathsf{class}\langle_,_,_\rangle \\
v.x & \text{where} & v \text{ is a value and } v \neq \{x_i = v_i\}^{i \in I} \text{ or } \forall i.x \neq x_i
\end{array}$$

Note that programs that would generate a *message-not-understood* error are faulty. Specifically, they contain selection of a record component not present in the record, since objects are records in our calculus.

Lemma 6.3 (Faulty Programs are Untypable) *If p is faulty there is no Γ, τ such that $\Gamma \vdash p : \tau$.* □

We are finally able to state and prove that our calculus is sound.

Theorem 6.4 (Soundness) *Let p be a program: if $\varepsilon \vdash p : \tau$ then either p diverges, or p evaluates to a and $\varepsilon \vdash a : \tau$, for some answer a.* □

The proof techniques are inspired by the ones of [31]. The complete set of definitions and properties with their proofs (omitted here for lack of space) may be found in [3].

7 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-based calculi:

- In the simplest of Cook's calculi [16], 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 in [25] 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 [30] 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. Another solution, proposed in [28], is to rename the superclass fields, but this does not ensure consistent initialization.
- TOOPL [8] 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 [13], 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 [12], 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 for 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 [23] for a complete account).

- Another approach to modeling classes as objects is developed by Fisher [20] 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 [22]: 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.
- Pierce and Turner [27] 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 [24] 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 λ-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 [18]. 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. [11] 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.

8 Conclusions and Future Work

We have presented a core calculus of classes and objects. This calculus is inherently class-based, *i.e.*, classes are the only object-oriented primitives. The main strengths of the calculus are its simplicity (the type system is an extension of a functional type system with record and reference types) and its power in modeling many attractive features of class-based languages. Our class construct provides a coherent, extensible collection of methods and fields, guarantees correct initialization of newly created objects, and automatically propagates superclass changes along the class hierarchy — all of these are desirable features for a formalism used to model classes [23].

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*).

Our calculus can be viewed as an extension of an ML-like language with subtypable records. In ML, all fundamental concepts such as functions, datatypes, tuples, etc. are expressed directly in the language, while in our calculus the concept of a *generator*, which is an essential part of the calculus, is related to the intermediate concept of a function from *self* to a record of methods (see section 4) which borrows records from the underlying language. This observation can lead to two different conclusions: (i) it may indicate that classes are a particular sort of functions, adding evidence to what has been pointed out by Cook [16]; (ii) it may suggest that the generator concept is just an intermediate point in the search for a good design for a class-based language.

In our design, we have been careful to minimize the dependencies between subclasses and superclasses: there is no interference between the private fields of the subclass and the superclass, object construction is modular, etc. The logical extension of this approach is to model subclasses that are parameterized over a family of superclasses. This direction is pursued in [7] which extends the calculus presented in this paper to model *mixin*-based inheritance.

References

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

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

[3] Bono, V., "Type Systems for the Object Oriented Paradigm," Ph.D. thesis, Università di Torino (1999).

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

[5] Bono, V. and K. Fisher, *An imperative, first-order calculus with object extension*, in: *Proc. ECOOP '98* (1998), pp. 462–497, preliminary version appeared in FOOL 5 proceedings.

[6] Bono, V. and L. Liquori, *A subtyping for the Fisher-Honsell-Mitchell lambda calculus of objects*, in: *Proc. CSL '94* (1995), pp. 16–30.

[7] Bono, V., A. Patel and V. Shmatikov, *A core calculus of classes and mixins*, in: *Proc. ECOOP '99*, 1999, to appear.

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

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

[10] Bruce, K. B., L. Cardelli, G. Castagna, T. H. O. Group, G. Leavens and B. C. Pierce, *On binary methods*, Theory and Practice of Object Systems **1** (1995), pp. 217–238.

[11] Bruce, K. B., L. Cardelli and B. C. Pierce, *Comparing object encodings*, in: *Proc. TACS '97* (1997), pp. 415–438.

[12] Bruce, K. B., L. Petersen and A. Finch, *Subtyping is not a good "match" for object-oriented languages*, in: *Proc. ECOOP '97* (1997), pp. 104–127.

[13] Bruce, K. B., A. Schuett and R. van Gent, *PolyTOIL: A type-safe polymorphic object-oriented language*, in: *Proc. ECOOP '95* (1995), pp. 26–51.

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

[15] Cook, W., W. Hill and P. Canning, *Inheritance is not subtyping*, in: *Proc. POPL '90*, 1990, pp. 125–135.

[16] Cook, W. R., "A Denotational Semantics of Inheritance," Ph.D. thesis, Brown University (1989).

[17] Crank, E. and M. Felleisen, *Parameter-passing and the lambda calculus*, in: *Proc. POPL '91*, 1991, pp. 233–244.

[18] Eifrig, J., S. Smith and V. Trifonov, *Sound polymorphic type inference for objects*, in: *Proc. OOPSLA '95*, 1995, pp. 169–184.

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

[20] Fisher, K., "Type Systems for Object-Oriented Programming Languages," Ph.D. thesis, Stanford University (1996).

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

[22] Fisher, K. and J. C. Mitchell, *A delegation-based object calculus with subtyping*, in: *Proc. 10th International Conference on Fundamentals of Computation Theory (FCT '95)* (1995), pp. 42–61.

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

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

[25] Kamin, S. 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 .

[26] Mason, I. and C. Talcott, *Programming, transforming, and proving with function abstractions and memories*, in: *Proc. ICALP '89* (1989), pp. 574–588.

[27] Pierce, B. C. and D. N. Turner, *Simple type-theoretic foundations for object-oriented programming*, J. Functional Programming **4** (1994), pp. 207–248, preliminary version appeared in Proc. *POPL '93* under the title *Object-Oriented Programming Without Recursive Types.*

[28] Reddy, U., *Objects as closures: Abstract semantics of object-oriented languages*, in: *Proc. Conference on Lisp and Functional Programming*, 1988, pp. 289–297.

[29] Stroustrup, B., "The C++ Programming Language (3rd ed.)," Addison-Wesley, 1997.

[30] Wand, M., *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 .

[31] Wright, A. and M. Felleisen, *A syntactic approach to type soundness*, Information and Computation **115** (1994), pp. 38–94.

A Definition of Contexts

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

$$
\begin{array}{rl}
C ::= & [\,] \mid C\ e \mid e\ C \mid \lambda x.C \mid C.x \\
& \mid \{m_1 = e_1, \ldots, m_{i-1} = e_{i-1}, m_i = C, m_{i+1} = e_{i+1}, \ldots, m_n = e_n\}^{1 \le i \le n} \\
& \mid \mathsf{H}\ h.C \mid \mathsf{H}\langle x, C\rangle h.e \mid \mathsf{new}\ C \mid \mathsf{class}\langle C, \mathcal{M}, \mathcal{P}\rangle \\
& \mid \begin{array}{l} \mathsf{extend}\ C\ \mathsf{with} \\ \quad \mathsf{method}\ m_i = v_{m_i}; \\ \quad \mathsf{protected}\ m_j = v_{m_j}; \\ \quad \mathsf{constructor}\ c; \\ \mathsf{end} \end{array}
\quad \mid \begin{array}{l} \mathsf{extend}\ e_s\ \mathsf{with} \\ \quad \mathsf{method}\ m_i = v_{m_i}; \\ \quad \mathsf{protected}\ m_j = v_{m_j}; \\ \quad \mathsf{constructor}\ C; \\ \mathsf{end} \end{array} \\
& \mid \begin{array}{l} \mathsf{extend}\ e_s\ \mathsf{with} \\ \quad \mathsf{method}\ m_i = v_{m_i};^{(i \in Pub \setminus [k])} \\ \quad \mathsf{method}\ m_k = C; \\ \quad \mathsf{protected}\ m_j = v_{m_j};^{(j \in Prot)} \\ \quad \mathsf{constructor}\ c; \\ \mathsf{end} \end{array}
\quad \mid \begin{array}{l} \mathsf{extend}\ e_s\ \mathsf{with} \\ \quad \mathsf{method}\ m_i = v_{m_i};^{(i \in Pub)} \\ \quad \mathsf{protected}\ m_j = v_{m_j};^{(j \in Prot \setminus [k])} \\ \quad \mathsf{protected}\ m_k = C; \\ \quad \mathsf{constructor}\ c; \\ \mathsf{end} \end{array}
\end{array}
$$

B Type Rules

The type rules for class-related forms in the core calculus 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.

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

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

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

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.

$$\frac{\mathit{typeof}(\mathit{const}) = \tau}{\Gamma \vdash \mathit{const} : \tau}\ (\mathit{const}) \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau}\ (\mathit{proj})$$

$$\frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x.e : \tau \to \sigma}\ (\lambda) \qquad \frac{\Gamma \vdash e_1 : \tau \to \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \sigma}\ (\mathit{app})$$

$$\frac{}{\Gamma \vdash fix : (\sigma \to \sigma) \to \sigma}\ (\mathit{fix}) \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau <: \sigma}{\Gamma \vdash e : \sigma}\ (\mathit{sub})$$

$$\frac{\Gamma \vdash e_i : \tau_i}{\Gamma \vdash \{x_i = e_i\}^{i \in I} : \{x_i : \tau_i\}}\ (\mathit{record}) \qquad \frac{\Gamma \vdash e : \{x : \sigma\}}{\Gamma \vdash e.x : \sigma}\ (\mathit{lookup})$$

$$\frac{}{\Gamma \vdash \mathsf{ref} : \tau \to \tau\ \mathsf{ref}}\ (\mathit{ref}) \qquad \frac{}{\Gamma \vdash\ ! : \tau\ \mathsf{ref} \to \tau}\ (!)$$

$$\frac{}{\Gamma \vdash :=\ : \tau\ \mathsf{ref} \to \tau \to \tau}\ (:=)$$

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