A Core Calculus of Classes and Objects

Viviana Bono

Dipartimento di Informatica
dell'Università di Torino
C.so Svizzera 185,
10149 Torino, Italy
bono@di.unito.it

Amit J. Patel Vitaly Shmatikov John C. Mitchell

Department of Computer Science
Stanford University
Stanford, CA 94305-9045
U.S.A.
{amitp,shmat,jcm}@cs.stanford.edu

November 19, 1998

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.

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

1 Introduction

While popular object-oriented languages such as C++ [Str97] and Java [Java96] 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 [CHC90, Bru94, BL95, FM95]. 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 main contribution of this paper is to show how advanced class-based language features can be modeled with a rather simple calculus that can be seen both as a proposal for a language design and as a step towards a foundational study of class-based languages. Our calculus 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 in our core calculus allows us to represent objects by records. As a consequence, the type system is simple, containing only functional types, record types, and reference types. In this setting, we need neither polymorphic record types, nor special class types.

We give design motivations and brief overview for 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.

2 Overview of the Core Calculus

In this section, we present our design motivations for a simple, sound calculus that is sufficiently expressive to support modern object-oriented programming techniques. We discuss various approaches 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 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, 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 which methods and fields are available only within the implementation of the class itself, which are available only to subclasses, and which are 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 the data representation is revealed – not even the number and names of fields. One of the benefits is that no conflicts arise if both the superclass and the subclass declare a private field of the same name.

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.

Modular object construction. Class hierarchies in a well-designed object-oriented program must not be fragile: if a superclass changes, 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 generator (sometimes also called constructor) of each class should provide the initial values only for the class's own fields, and call a superclass generator which will provide the initial values for the superclass fields and call its own superclass generator, etc. (this concept is illustrated by C++ constructors). Unlike many theoretical calculi for object-oriented languages, our calculus directly supports modular object construction.

2.2 Calculi for Object-Oriented Languages

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 [CW85], in which inheritance was modeled by record subtyping. Unlike records, however, objects' methods should be able to modify fields and invoke sibling methods [Cook89]. 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 modified when a method is inherited, since new methods and fields may be added to the host object.

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 fix-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, [AC96] 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. With late binding, the fix-point operator has to be applied at each method invocation.

Even more complicated is the case when the calculus supports object extension (*i.e.*, addition of new methods and fields) in a functional setting [FHM94, BSv95]. When an object is extended, the type given to self in the old methods has to be modified since the host object now contains additional methods and fields. Therefore, the type system should include the notion of *MyType* (also called *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.

Our goal is to achieve a reasonable trade-off between expressivity and simplicity. We do not support functional updates, because we believe that imperative updates combined with early self binding provide such a trade-off. Without functional updates, we can use early binding of self. Early binding eliminates the main need for recursive object types. 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 [FM95, BSv95]. It is also worth noting that popular statically typed object-oriented languages such as C++ and Java are imperative and do not support *MyType*.

There are, however, at least two possible drawbacks to our approach. Binary methods cannot be simulated without support for *MyType* (even though methods that return modified self can be modeled in our calculus as imperative methods modifying the object in place and returning *unit* type). 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

In the core calculus, classes are represented by *class values* (class expression forms). A class value is a tuple containing the generator function, the set of public method names, and the set of protected method names. A generator produces a function from self to the record of methods. When the class is instantiated, a fix-point operator is applied to the generator's result to bind self in the methods' bodies, creating a full-fledged object.

As mentioned in section 2.1, a class has two uses: inheritance and instantiation. Each of these has a corresponding expression in the core calculus: extend *expr* with ... is used for inheritance and new *expr* is

Expressions:

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

Values:

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

Figure 1: Syntax of the core calculus

used for instantiation.

The extend expression models inheritance by defining one class (subclass) as an extension of another (superclass). Essentially, 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 new expression, when evaluated, produces a new object from the class value by invoking the class generator and applying the fix-point operator to its result.

Objects being records in our calculus, object types are treated in the same way as record types in ML, except that subtyping is allowed on object types. Subtyping relations are 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 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 public or protected accessor methods.

Rather than putting encapsulation levels into object types, they are expressed 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 in the object type at all, but are instead bound in each method body. For simplicity, the core calculus allows exactly one private field per class. Each method body takes the class's private field as a parameter which is bound when the class generator constructs a new object.

3 Syntax of the Core Calculus

Our core 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.

The syntax of our calculus was inspired by that of *Reference ML* [WF94], where Wright and Felleisen analyze the operational soundness of a version of ML extended 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.

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 $const \in Const$; $x, x_i, m_i, m_j \in Var$; *fix* is the fix-point operator; ref , !, := are operators;[1] $\{x_i = e_i\}^{i \in I}$ is a record; h is a set of pairs $h ::= \{\langle x, v\rangle^*\}$ where $x \in Var$ and v is a value (first components of the pairs are all distinct); v_g is a value (a λ-abstraction); $[m_i]$, $[m_j]$ are sets of identifiers; v_c is a value (a λ-abstraction); and $I, Pub, Prot \subset \mathcal{NAT}$.

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 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:

- $\mathsf{class}\langle v_g, [m_i]^{i \in Pub}, [m_j]^{j \in Prot}\rangle$ 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.
- $\mathsf{extend}\ e_s\ \mathsf{with}$
  $\mathsf{public}\ m_i = e_{m_i};\ ^{(i \in Pub)}$
  $\mathsf{protected}\ m_j = e_{m_j};\ ^{(j \in Prot)}$
  $\mathsf{constructor}\ v_c;$
  end

  is a *class extension*, in which e_s represents the superclass, $m_i = e_{m_i}$ are public method declarations, and $m_j = e_{m_j}$ are protected method declarations. Each method body expression $e_{m_{i,j}}$ 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 $e_{m_{i,j}}$ 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 expresson, the v_c value is used to build the generator function v_g as described in section 4.
- new e v uses the generator v_g of the class value to which e evaluates to create a new object. The argument v is used to initialize the new object's fields as described in section 4.
- $e.x$ selects a record component.

Programs and *answers* are defined as follows:

$$\begin{array}{lll} p ::= & e_c & \text{where } e_c \text{ is a closed expression} \\ a ::= & v \mid \mathsf{H}\ h.v & \end{array}$$

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

$$\mathsf{Object} \triangleq \mathsf{class}\ \langle\ \lambda_.\lambda_.\{\},\ [\ ],\ [\ ]\ \rangle$$

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 $(\lambda x.e_2)e_1$, and we use $e_1 ; e_2$ as an abbreviation for $(\lambda_.e_2)e_1$.[2] Also, we use unit as an abbreviation for

[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 [WF94], this is just a syntactic convenience, as is the curried version of :=. Standard forms used in the examples are the syntactic sugar of the presented syntax.

[2] This works since we are in a call-by-value setting.

$$
\begin{array}{rcll}
(\lambda x.e)\ v & \rightarrow & [v/x]\ e & (\beta_v) \\
\mathit{fix}\ (\lambda x.e) & \rightarrow & [\mathit{fix}\,(\lambda x.e)/x]e & (\text{fix}) \\
\{\ldots, x = v, \ldots\}.x & \rightarrow & v & (\text{select}) \\
\mathsf{ref}\ v & \rightarrow & \mathsf{H}\langle x, v\rangle.x & (\text{ref}) \\
\mathsf{H}\langle x, v\rangle h.R[!x] & \rightarrow & \mathsf{H}\langle x, v\rangle h.R[v] & (\text{deref}) \\
\mathsf{H}\langle x, v\rangle h.R[:= x v'] & \rightarrow & \mathsf{H}\langle x, v'\rangle h.R[v'] & (\text{assign}) \\
R[\mathsf{H}\ h.e] & \rightarrow & \mathsf{H}\ h.R[e], \quad R \neq [\,] & (\text{lift}) \\
\mathsf{H}\ h.\mathsf{H}\ h'.e & \rightarrow & \mathsf{H}\ h\ h'.e & (\text{merge}) \\
\mathsf{new\ class}\langle g, \mathcal{M}, \mathcal{P}\rangle\ v & \rightarrow & \mathit{Sub}_{\mathcal{M}\cup\mathcal{P}\rightarrow\mathcal{M}}(\mathit{fix}\,(g\ v)) & (\text{new}) \\
\left(\begin{array}{l}\mathsf{extend\ class}\langle g_s, \mathcal{M}_s, \mathcal{P}_s\rangle\ \mathsf{with} \\ \quad \mathsf{public}\ m_i = e_{m_i};\ {}^{(i\in Pub)} \\ \quad \mathsf{protected}\ m_j = e_{m_j};\ {}^{(j\in Prot)} \\ \quad \mathsf{constructor}\ c; \\ \mathsf{end}\end{array}\right) & \rightarrow & \mathsf{class}\langle \mathit{Gen}, [m_i]\cup\mathcal{M}_s, [m_j]\cup\mathcal{P}_s\rangle & (\text{class}) \\
 & & (\text{where } \mathit{Gen} = \lambda x.\lambda \text{self}.\cdots \text{ as defined below}) &
\end{array}
$$

Figure 2: Reduction Rules

the empty record or type $\{\}$, instead of having a new *unit* value and type as in ML. 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.

An example

To illustrate the core calculus, we present a simple example that demonstrates the syntax of a class extension, the binding of self, private field, and next in methods, and the use of records in the constructor components.

```
let A =                                   let B =
   extend Object with                        extend A with
    method m = λa. λself. λx. x+a             method m = λnext. λb. λself. λx.
    constructor λz. {fieldinit=z, superinit={}}                 next(x) * b
   end in                                     constructor λy. {fieldinit=y, superinit=17}
                                           end in ...
```

Method m is a function that takes a private field (a in class A and b in class B), next if m is redefining a superclass method with the same name, and self, a reference to the host object containing m. Note that class B's methods can only access the private field b of B and not the private field a of A.

The constructor component of extend is used to build the class generator v_g when class extension expression is evaluated (see section 4). It is a function that takes any argument and returns a record with two fields, fieldinit and superinit. The fieldinit value is used to initialize the class's private field. The superinit value is used as an argument to the superclass generator.

4 Operational Semantics

The operational semantics for our calculus extends that of *Reference* ML [WF94]. Reduction rules are given in Fig.2, where R are *reduction contexts* [CF91, FH92, MT89]. 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 are defined as follows:

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

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 [WF94]: (*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 location it is looking for in the closest local heap.

Rule (*class*) evaluates class extension expressions. An extend expression (which can be seen intuitively as a class declaration, where $\mathsf{class}\langle g_s, \mathcal{M}_s, \mathcal{P}_s\rangle$ represents its superclass) is reduced to a triple $\langle Gen, [m_i] \cup \mathcal{M}_s, [m_j] \cup \mathcal{P}_s\rangle$ where Gen is the generator function defined below, $[m_i] \cup \mathcal{M}_s$ is the set of public method names, and $[m_j] \cup \mathcal{P}_s$ is the set of protected method names. $\mathcal{M}_s$ and $\mathcal{P}_s$ are respectively the public and protected methods of the superclass. Using a class generator delays full inheritance resolution until object instantiation time, when self is available.

We define $\mathcal{A}_s = \mathcal{M}_s \cup \mathcal{P}_s$ 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 of extend to initialize the field of the new object, and returns a function from self to a record of methods. When the fix-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).

$$
\begin{array}{l}
Gen \triangleq \lambda x.\lambda \mathrm{self}.\mathsf{let}\ \ t = c(x) \quad \mathsf{in} \\
\qquad \mathsf{let}\ \ \mathrm{field} = t.\mathrm{fieldinit} \quad \mathsf{in} \\
\qquad\quad \mathsf{let}\ \ \mathrm{superobj} = g_s(t.\mathrm{superinit})\ \mathrm{self} \quad \mathsf{in} \\
\qquad\qquad \left\{ \begin{array}{l} m_k = e_{m_k}\ \mathrm{field\ self} \\ m_l = \mathrm{superobj}.m_l \\ m_r = e_{m_r}\ \mathrm{superobj}.m_r\ \mathrm{field\ self} \end{array} \right\}^{m_k \in \mathcal{A}_e \setminus \mathcal{A}_s, m_l \in \mathcal{A}_s \setminus \mathcal{A}_e, m_r \in \mathcal{A}_e \cap \mathcal{A}_s}
\end{array}
$$

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, the other is the superclass generator's argument. Gen first computes the initial value of the field by evaluating $t.\mathrm{fieldinit}$, where $t = c(x)$. Gen then calls the superclass generator g_s (passing argument $t.\mathrm{superinit}$) to obtain a function from self to a record of superclass methods. The record computed as a result of applying this function to self we call $\mathrm{superobj}$. Note that the definition of Gen makes sense if c and g_s are defined appropriately, that is, they behave as expected if well-typed (see Section 5).

Finally, Gen builds a function from self that returns a record containing *all* methods — from both the subclass and the superclass. To understand how the record is created, recall that method bodies take parameters field, self, and, if redefined, next . 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 has to bind field for them. Methods $m_l \in \mathcal{A}_s \setminus \mathcal{A}_e$ are the inherited superclass methods: they are taken intact from superobj. 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 e_{m_r} from the subclass definition), but it can still refer to the old method via the next parameter, which is bound to $\text{superobj}.m_r$ by Gen.

The definition of Gen presented here has been simplified for explanation purposes. In particular, this version has the technical drawbacks that it does not behave properly in the presence of side effects (method bodies may be evaluated more than once when the fix-point operator is applied to the function returned by Gen, and they potentially contain side effects), nor does it work perfectly with the rule (*fix*). The complete (and correct) definition of Gen can be found in appendix C. That one behaves properly in the presence of side effects and is consistent with the (*fix*) reduction rule.

Note that we have a rule for class extensions but not for the root Object class; we defined Object to be a class value, so it does not need to be reduced.

Rule (*fix*) can be seen as similar to the more standard reduction rule for the μ operator

$$\mu x.e \;\rightarrow\; [\mu x.e/x]e.$$

We decided to use the constant *fix* to keep the whole calculus simpler. Moreover, a version of the system where the call-by-value fix-point operator Y_v of [Plo75] is used can be found in [Bo98]. We chose not to present the version of the calculus with Y_v in order to keep the technical definition of Gen simple; the two versions are equivalent in terms of expressive power.

Rule (*new*) creates a new object. It first applies generator g to argument v,[3] creating a function from self to a record of methods. Then the fix-point operator *fix* (following [Cook89]) is applied to bind self in method bodies and create a recursive record. Finally, $Sub_{\mathcal{M}\cup\mathcal{P}\rightarrow\mathcal{M}}$ is a coercion function from records to records that hides all components which are in $\mathcal{M} \cup \mathcal{P}$ but not in $\mathcal{M}$. The resulting record contains only public methods, and can be returned to the user as a fully formed object.

Now it should be clear why we need to carry method names in a class value. They are used in Gen to distinguish which methods are inherited intact from the superclass and which are redefined by the subclass.

The operational semantics presented in this section is fairly straightforward and requires no special technical machinery, yet provides support for data encapsulation and modular object construction.

5 Type System

Our types are standard even though we have to handle class expressions (class, extend and new), and the typing rules are fairly straightforward. Types are as follows:

$$\tau ::= \quad \iota \mid \tau_1 \rightarrow \tau_2 \mid \{x_i : \tau_i\}^{i\in I} \mid \mathsf{class}\langle \tau, \{m_i{:}\tau_i\}^{i\in Pub}, \{m_j{:}\tau_j\}^{j\in Prot}\rangle \mid \tau \text{ ref}$$

where ι is a constant type; $\rightarrow$ 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 ($Pub \cap Prot = \emptyset$); $I, Pub, Prot \subset \mathcal{NAT}$; and τ ref

[3] It is not necessary for the argument to be present to reduce a new expression, but we include it to make the presentation easier to follow.

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

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

$$\frac{\begin{array}{lll} \text{(Parent)} & \text{For } i \in Pub_s, j \in Prot_s: & \Gamma \vdash e_s : \mathsf{class}\langle \rho, \{m_i{:}\tau_{m_i}\}, \{m_j{:}\tau_{m_j}\} \rangle \\ \text{(New Meths)} & \text{For } i \in A_e \setminus A_s: & \Gamma \vdash e_{m_i} : \eta \to \sigma \to \sigma_{m_i} \\ \text{(Redef Meths)} & \text{For } i \in A_e \cap A_s: & \Gamma \vdash e_{m_i} : \tau_{m_i} \to \eta \to \sigma \to \sigma_{m_i} \qquad \Gamma \vdash \sigma_{m_i} <: \tau_{m_i} \\ \text{(Constr Comps)} & & \Gamma \vdash c : \gamma \to \{\text{fieldinit} : \eta, \text{superinit} : \rho\} \end{array}}{\Gamma \vdash \left(\begin{array}{l} \mathsf{extend}\ e_s\ \mathsf{with} \\ \quad \mathsf{public}\ m_i = e_{m_i};\ ^{(i \in Pub_e)} \\ \quad \mathsf{protected}\ m_j = e_{m_j};\ ^{(j \in Prot_e)} \\ \quad \mathsf{constructor}\ c \\ \mathsf{end} \end{array} \right) : \mathsf{class}\langle \gamma, \{m_i{:}\sigma_{m_i}\}^{i \in (Pub_s \cup Pub_e)}, \{m_j{:}\sigma_{m_j}\}^{j \in (Prot_s \cup Prot_e)} \rangle} \quad (\textit{extend})$$

where: $A_s = Pub_s \cup Prot_s$ $\quad A_e = Pub_e \cup Prot_e$ $\quad A = A_s \cup A_e$
$\sigma_{Pub} = \{m_i : \sigma_{m_i}\}^{m_i \in Pub_s \cup Pub_e}$ $\quad \sigma_{Prot} = \{m_j{:}\tau_{m_j}\}^{j \in Prot_s \cup Prot_e}$ $\quad \sigma = \{m_k : \sigma_{m_k}\}^{k \in A}$

Figure 3: Typing Rules for Class-Related Forms

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.

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.

Our 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 judgements are as follows:

$\Gamma \vdash \tau_1 <: \tau_2$ $\quad$ τ_1 is a subtype of τ_2
$\Gamma \vdash e : \tau$ $\quad$ e has type τ.

The set of typing rules for class-related forms is shown in Fig.3. The remaining rules are shown in appendix B. The rules are fairly standard since our calculus extends a (non-polymorphic) λ-calculus with references. Simplicity is the most appealing feature of our type system.

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 a generator which produces a function that will later (at the time of new application) return a real object. The type of g (see

section 4 for the formal definition of the generator) can be determined by examining the type of a 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 fix-point operator is applied. The return type of g is therefore $\sigma \to \sigma$, where σ represents the type of self, $\{m_k : \tau_k\}^{k\in Pub\cup Prot}$ ($\mathrm{fix}(g(v))$ has type σ). This record type includes *all* methods, not only public methods.

Rule (*extend*) is relatively long but straightforward. We describe it following the order of the subexpressions in the conclusion of the rule:

- The superclass expression e_s is expected to be typed as a class, where ρ is the type of the superclass generator argument, and the two sets of names are, respectively, the names of the public and protected superclass methods.
- The bodies of the new methods are typed with a function type (see the second premise of the rule). 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 type.
- The bodies of the redefined methods are also typed with a function type (see the third premise of the rule). 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 above. Redefined method bodies may have a more specific type than the old ones (premise $\Gamma \vdash \sigma_{m_i} <: \tau_{m_i}$).
- $\mathsf{constructor}$ expression c is a function that returns a record of two expressions. $\mathrm{fieldinit}$ is the initialization expression for the private field. Clearly, it has to have the same type η as the one assumed for the field when typing methods bodies. $\mathsf{superinit}$ is the expression passed as the actual argument to the superclass generator. It has to have the same type ρ as the parameter of the superclass generator. Recall that because of encapsulation, a class in our core 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.

To better understand why this type includes all the relevant data about the class, consider class values (class expressions) which are typed by the (*class val*) rule described above. Their type is essentially identical to that given to extend expressions, which makes sense given that extend expressions evaluate to class values (see section 4).

For Object in (*root class*) rule, the generator takes nothing (*i.e.*, unit) as a parameter. The resulting object has no methods, so the two sets of method names are empty. All classes other that Object are generated by an extend expression, which takes an expression with a class type representing a superclass and extends it to build a subclass. When a class is extended, the new method names are added to the old method names to produce the set of method names for the new class value.

Rule (*instantiate*) types the creation of a new object. The $\mathsf{new}\ e$ term is typed as a function that takes the generator's argument and returns a fully initialized object.

In our calculus, objects are represented by records, thus method invocation is simply record component selection, as the rule (*lookup*) shows.

5.1 The Core Calculus is Sound

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

Lemma 5.1 (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 a way to approximate the concept of reaching 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 5.1 (Faulty Programs) *The* faulty *programs are the ones containing some of the following sub-expressions:*

cv	*where*	*c is a constant, v a value but $\delta(c, v)$ is not defined;*
$v_1 v_2$	*where*	*v_1, v_2 are values and $v_1 \neq \lambda x.e, \mathsf{ref}, !, :=, := v$;*
$!v$	*where*	*v is a value and $v \neq x$;*
$:= v$	*where*	*v is a value and $v \neq x$;*
$\mathsf{H}h\langle x, v_2\rangle.C[xv_1]$	*where*	*v_1, v_2 are values;*
$\mathsf{new}\ v$	*where*	*v is a value and $v \neq \mathsf{class}\langle _, _, _\rangle$;*
$\mathsf{extend}\ v\ \mathsf{with} \ldots \mathsf{end}$	*where*	*v is a value and $v \neq \mathsf{class}\langle _, _, _\rangle$;*
$v.x$	*where*	*v is a value and $v \neq \{x_i = v_i\}^{i \in I}$ or $x \neq x_i, \forall i$.*

It is worth noting that faulty programs contain those with the so called *message-not-understood* error. The corresponding faulty program is the selection of a record component not present in the record, since objects are records in our calculus.

Lemma 5.2 (Faulty Programs are Untypable) *If p is faulty there are no Γ, τ such that $\Gamma \vdash p : \tau$.*

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

Theorem 5.3 (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 [WF94]. The complete set of definitions and properties with their proofs (omitted here for lack of space) may be found in [Bo98].

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-based calculi:

- In the simplest of Cook's calculi [Cook89], objects are represented by records of methods, and created by taking the fix-point of the function representing the class (*constructor* in Cook's terminology). Just as in our calculus, self is bound early. Inheritance is modeled by generating the subclass constructor from the superclass constructor. However, classes are not a basic construct. The calculus relies on record concatenation operators, and typing issues are not resolved.

- The calculus of Wand in [Wand94] 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. Objects are records. Like in our calculus, Wand's calculus has early self binding. The new operation (Wand calls it *constructor*) is an application of the fix-point operator. For our calculus, we showed in section 4 that new operation is an application of the fix-point operator as well. Our calculus also has features not present in Wand's calculus, such as modular object creation. Wand models inheritance in such a way that the subclass must know and directly initialize the fields of the superclass. In our calculus, modularity is achieved by having each class's field initialized by the class's own generator. Another solution, proposed in [Re88], is to rename the superclass fields, but this does not ensure consistent initialization.
- *TOOPL* [Bru93] 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* [BSv95], where one of the most remarkable things is the complete separation between subtyping and inheritance. Different relations are used for the two: 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 [BPF97], another language is presented, *Loom*, where only matching is used and the type system is simplified.

Our core calculus is an attempt to built 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 [AC96]. 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 and 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 to some basic requirements for class constructs (see section 2 above, and [FM98] for a complete account).
- Another proposal to modeling classes as objects is presented in [FM98] and formalized in [BF98] (with an imperative calculus). Classes are modeled as encapsulated extensible objects; inheritance is modeled as an extension operation on objects. Following [FM95], an object can be in one of two states: a *prototype* (can be extended, but subtyping does not apply), and a "proper" subtypable object. Existential quantification is used to (partially) abstract the class implementation.
- In [PT94], Pierce and Turner 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. Also, binary methods require extra machinery since their model provides encapsulation at the object rather than class level. One solution appears in [PT93]. Another solution is proposed in [HP96]: Hofmann and Pierce 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 setting, we do not put encapsulation at the object type level. Instead, we use subtyping to hide protected methods and binding to hide private fields. We believe this could be considered a rather simple but effective way to obtain all the encapsulation needed in practice.

- Scott Smith and the Hopkins Object Group have designed a type-safe class-based object-oriented language with a rich feature set called I-LOOP [EST95]. 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. Currently, the main problem is that inferred types are large and difficult to read. Work in this area is in progress.

There is an interesting paper [BCP97] which shows 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 our core calculus might be a step in this direction.

7 Conclusions and Future Work

We have presented a core calculus of classes and objects. This calculus is totally and inherently class-based, *i.e.*, the only object-oriented related primitives are classes. It is possible to think about a straightforward two-level (an *implementor* level and a *user* level) version of the calculus, where records are not available to users. In this way, objects may be generated only from classes. 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 a class-based language. 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 [FM98].

Some of the design choices may appear debatable, in particular 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 solely by efficiency reasons, and, in fact, the calculus can be easily extended to support *super* by keeping the entire $\mathrm{superobj}$ instead of selecting its components each time (see Section 4).

Our calculus can be viewed as an extension of an ML-like language with subtypable records and references. 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), that 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 Cook [Cook89] and others have already pointed out; (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.

All in all, 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 could be the starting point for more foundational studies such as denotational semantics for the class construct.

Following are the topics for future research:

- Extend the core calculus to model parameterized inheritance in addition to conventional single inheritance. An extension of the core calculus that supports *mixins* is a work in progress.
- Develop an efficient implementation for the core calculus, and then extend it to a full language.
- Study an extension of the core calculus with ML polymorphism in order to be able to combine classes and objects with the full power of ML type inference.
- Consider combining existential types with our simple object types to provide a form of implementation types. Implementation types would be subtypes of their corresponding interface types.

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer–Verlag, 1996.

[BCP97] K. Bruce, L. Cardelli and B. C. Pierce. Comparing Objects Encodings. In *TACS '97*, *LNCS* 1281, 415–438, Springer-Verlag, 1997.

[BF98] V. Bono and K. Fisher. An Imperative First-Order Calculus with Object Extension. In *FOOL '98*.

[BL95] V. Bono and L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In *CSL '94*, *LNCS* 933, 16-30. Springer-Verlag, 1995.

[Bo98] V. Bono. Type Systems for the Object Oriented Paradigm. PhD thesis, work in progress. `http://www.di.unito.it/~bono`

[BPF97] K.B. Bruce, L. Petersen and A. Finch. Subtyping is not a Good "Match" for Object-Oriented Languages. In *ECOOP '97*, *LNCS* 1241, 104–127. Springer-Verlag, 1997.

[Bru93] K. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *POPL '93*, pages 285–298. ACM Press, 1993.

[Bru94] K.B. Bruce. A Paradigmatic Object–Oriented Programming Language: Design, Static Typing and Semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[BSv95] K. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *ECOOP '95* , *LNCS* 952, 26–51. Springer-Verlag, 1995.

[Car88] L. Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, 76:138–164, 1988.

[CF91] E. Crank and M. Felleisen. Parameter-passing and the lambda calculus. In *POPL '91*, 233-234. ACM Press, 1991.

[CHC90] W. Cook, W. Hill, and P. Canning. Inheritance is not Subtyping. In *POPL '90*, 125–135. ACM Press, 1990.

[CM91] L. Cardelli and J.C. Mitchell. Operations on Records. *Mathematical Structures in Computer Sciences*, 1(1):3–48, 1991.

[Cook89] W.R. Cook. A Denotational Semantics of Inheritance. PhD thesis, Brown University, 1989.

[Cop92] J. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *OOPSLA '95*, 169–184. ACM Press, 1987.

[FH92] M. Felleisen and R. Hierb. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, 102:235–271, 1992.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994. Preliminary version appeared in *LICS '93*, 26-38. IEEE Press, 1993.

[Fis96] K. Fisher. Type Systems for object-oriented programming languages. PhD thesis, Stanford University, 1996.

[FM95] K. Fisher and J.C. Mitchell. A delegation-based object calculus with subtyping. In *10th Int'l Conf. Fundamentals of Computation Theory (FCT '95)*, *LNCS* 965, 42–61. Springer-Verlag, 1995.

[FM98] K. Fisher and J.C. Mitchell. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems*, 1998. To Appear. Preliminary version appeared in Marktoberdorf '97 proceedings.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80, The Language and its Implementation*. Addison Wesley, 1983.

[Ha94] R. Harper. A Simplified Account of Polymorphic References. In *Infomation Processing Letters*, 51(4):201–206, 1994.

[HP96] M. Hofmann and B.C. Pierce. Positive Subtyping. *Information and Computation*, 126(1):11–33, 1996. Preliminary version appeared in *POPL '95*. ACM Press, 1995.

[Java96] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[MT89] I. Mason and C. Talcott. Programming, transforming, and proving with function abstractions and memories. In *ICALP '89*, *LNCS* 372, 574-588. Springer-Verlag, 1989.

[Plo75] G. Plotkin. Call-by name, Call-by-value and the Lambda-Calculus. *Theoretical Computer Science*, 1:125-159, 1975.

[PT93] B.C. Pierce and D.N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[PT94] B.C. Pierce and D.N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, 1994. Preliminary version appeared in POPL '93, under the title "Object-oriented programming without recursive types".

[Re88] U. Reddy. Objects as Closures: Abstract Semantics of Object-Oriented Languages. In *Conference on Lisp and Functional Programming*, 289–297. ACM Press, 1988.

[Re95] D. Rémy. From Classes to Objects via Subtyping. Draft, INRIA, 1997.

[Sny86] A. Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *OOPSLA '86*, –. ACM Press, 1986.

[Str97] B. Stroustrup. *The C++ Programming Language (3rd ed.)*. Addison-Wesley, 1997.

[To90] L. Thorup. Type Inference for Polymorphic References. In *Information and Computation*, 89(1):1–34, 1990

[US87] D. Ungar and R. B. Smith. Self: the Power of Simplicity. In *OOPSLA '87*, 227–241. ACM Press, 1987.

[W87] M. Wand. Complete Type Inference for Simple Objects. In *LICS '87*, 37–44. IEEE Press, 1987.

[Wand94] M. Wand. Type Inference for Objects with Instance Variables and Inheritance. In *Theoretical Aspects of Object- Oriented Programming: Types, Semantics, and Language Design*. MIT Press, 1994.

[WF94] 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 extend expression:

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

$$
\begin{array}{ll}
\mid \begin{array}{l}\mathsf{extend}\ C\ \mathsf{with} \\ \quad \mathsf{public}\ m_i = e_{m_i}; \\ \quad \mathsf{protected}\ m_j = e_{m_j}; \\ \quad \mathsf{constructor}\ c; \\ \quad \mathsf{end}\end{array} & \mid \begin{array}{l}\mathsf{extend}\ e_s\ \mathsf{with} \\ \quad \mathsf{public}\ m_i = e_{m_i}; \\ \quad \mathsf{protected}\ m_j = e_{m_j}; \\ \quad \mathsf{constructor}\ C; \\ \quad \mathsf{end}\end{array} \\
\mid \begin{array}{l}\mathsf{extend}\ e_s\ \mathsf{with} \\ \quad \mathsf{public}\ m_i = e_{m_i};^{(i \in Pub \setminus [k])} \\ \quad \mathsf{public}\ m_k = C; \\ \quad \mathsf{protected}\ m_j = e_{m_j};^{(j \in Prot)} \\ \quad \mathsf{constructor}\ c; \\ \quad \mathsf{end}\end{array} & \mid \begin{array}{l}\mathsf{extend}\ e_s\ \mathsf{with} \\ \quad \mathsf{public}\ m_i = e_{m_i};^{(i \in Pub)} \\ \quad \mathsf{protected}\ m_j = e_{m_j};^{(j \in Prot \setminus [k])} \\ \quad \mathsf{protected}\ m_k = C; \\ \quad \mathsf{constructor}\ c; \\ \quad \mathsf{end}\end{array}
\end{array}
$$

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.

$$\frac{}{\Gamma, \iota_1 <: \iota_2 \vdash \iota_1 <: \iota_2} \quad (<: \textit{ proj}) \qquad \frac{}{\Gamma \vdash \tau <: \tau} \quad (\textit{refl}) \qquad \frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3} \quad (\textit{trans})$$

$$\frac{\Gamma \vdash \tau' <: \tau \quad \Gamma \vdash \sigma <: \sigma'}{\Gamma \vdash \tau \to \sigma <: \tau' \to \sigma'} \quad (\textit{arrow}) \qquad \frac{\Gamma \vdash \tau_i <: \tau_j \quad i \in I, j \in J, I \subseteq J}{\Gamma \vdash \{m_i : \tau_i\}^{i \in I} <: \{m_j : \tau_j\}^{j \in J}} \quad (\textit{rec})$$

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.

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

$$\frac{\Gamma \vdash e_1 : \tau \to \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\, e_2 : \sigma} \quad (\textit{app}) \qquad \frac{}{\Gamma \vdash fix : (\sigma \to \sigma) \to \sigma} \quad (\textit{fix})$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau <: \sigma}{\Gamma \vdash e : \sigma} \quad (\textit{sub}) \qquad \frac{\tau \textit{ type}}{\Gamma \vdash \mathsf{ref} : \tau \to \tau\ \mathsf{ref}} \quad (\textit{ref}) \qquad \frac{\tau \textit{ type}}{\Gamma \vdash\ !\, : \tau\ \mathsf{ref} \to \tau} \quad (!)$$

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

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

C Formal Definition of Gen

Here we present the complete, formal definition of the Gen expression introduced in section 4, which works correctly at fix-point application time. This version differs from the one in section 4 in that the method definitions are evaluated exactly once per class (by lifting them outside the λx binding), the value of the field is evaluated once per object (by lifting it outside the λself binding), the superclass generator is called once per object (also by lifting it outside λself), and the methods (including generation of the superclass methods) are enclosed in an η-expansion to avoid non-termination when reducing the fix-point of the λself portion of Gen.

$$\begin{array}{l}
Gen \triangleq \mathsf{let}\ \mathrm{meth} = \{m_k = e_{m_k}, m_r = e_{m_r}\}^{m_k \in \mathcal{A}_e \setminus \mathcal{A}_s, m_r \in \mathcal{A}_e \cap \mathcal{A}_s} \quad \mathsf{in} \\
\quad \lambda\, x. \\
\quad\quad \mathsf{let}\ t = c(x) \quad \mathsf{in} \\
\quad\quad\quad \mathsf{let}\ \mathrm{field} = t.\mathrm{fieldinit} \quad \mathsf{in} \\
\quad\quad\quad\quad \mathsf{let}\ \mathrm{supergen} = g_s(t.\mathrm{superinit}) \quad \mathsf{in} \\
\quad\quad\quad\quad\quad \lambda\, \mathrm{self}. \\
\quad\quad\quad\quad\quad\quad \left\{ \begin{array}{l} m_k = \lambda y.\ \mathrm{meth}.m_k\ \mathrm{field}\ \mathrm{self}\ y, \\ m_l = \lambda y.\ (\mathrm{supergen}\ \mathrm{self}).m_l\ y, \\ m_r = \lambda y.\ \mathrm{meth}.m_r\ (\mathrm{supergen}\ \mathrm{self}).m_r\ \mathrm{field}\ \mathrm{self}\ y \end{array} \right\}^{m_k \in \mathcal{A}_e \setminus \mathcal{A}_s, m_l \in \mathcal{A}_s \setminus \mathcal{A}_e, m_r \in \mathcal{A}_e \cap \mathcal{A}_s}
\end{array}$$