Objective ML:
A simple object-oriented extension of ML

Didier Rémy and Jérôme Vouillon*

Abstract

Objective ML is a small practical extension of ML with objects and toplevel classes. It is fully compatible with ML; its type system is based on ML polymorphism, record types with polymorphic access, and a better treatment of type abbreviations. Objective ML allows for most features of object-oriented languages including multiple inheritance, methods returning self and binary methods as well as parametric classes. This demonstrates that objects can be added to strongly typed languages based on ML polymorphism.

Introduction

We propose a simple extension of ML with class-based objects. *Objective ML* is a fully conservative extension of ML. A beginner may ignore the object extension. Moreover, he would not notice any difference, even in the types inferred. This is possible since the type inference algorithm of Objective ML, as in ML, is based on first-order unification. Types are extended with object types that are similar to record types for polymorphic access. It is also essential to improve both the status and the treatment of type abbreviations in order to keep types readable.

When using object-oriented features, the user is never required to write interfaces of classes, although he might have to include a few type annotations when defining parametric classes or coercing objects to their counterparts in super classes.

Objective ML is a class-based system. Objects are records of methods. Our language copes with most features of object-oriented programming, including methods returning self, binary methods, virtual classes and multiple inheritance. Coercion from objects to their counterparts in super classes is also possible but must be written explicitly. Classes expose the states of objects to their subclasses, but the states are hidden in the objects themselves. Both the status and the treatment of type abbreviations have been improved so that types are easier to read.

The ingredients used, except automatic abbreviations, are not new. However, their incorporation into a practical language, combining power, simplicity and compatibility with ML, is new.

Objective ML is formally defined, and its dynamic semantics is proven correct with respect to the static semantics. The language has not been designed to be a minimal calculus of objects, but rather the core of a real programming language. In particular, the semantics of classes is compatible with programming in imperative as well as functional style and it allows for efficient memory management (methods can be shared between all the instances of a class).

The rest of this paper is organized as follows: the first section is an overview of Objective ML. Objects and classes are introduced in sections 2 and 3, respectively. Coercions are dealt with in section 4. The semantics of the language is described in section 5. Type inference is discussed in section 6. The abbreviation mechanism is explained in section 7. In section 9, we compare our proposal with other works.

1 An overview of Objective ML

Objective ML has been implemented on top of the Caml Special Light system [Ler96]. We have used this implementation, now called Objective Caml[1], to process all examples shown below. When useful, we display the output of the typechecker in a slanted font. Toplevel

*Authors' address: INRIA-Rocquencourt, B.P. 105, F-78153 Le Chesnay Cedex, France.
Email: Didier.Remy,Jerome.Vouillon@inria.fr

To appear in the proceedings of the 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, January 15–17, 1997.

[1] The syntax has been slightly modified here in order to keep the concrete syntax and the abstract syntax closer.

definitions are implicit `let ... in ...`. For each phrase, the typechecker outputs the binding that will be generalized and added to the global environment before typechecking the next phrase.

The language Objective ML is class-based. That is, objects are usually created from classes, even though it is also possible to create them directly (this is described in the next section). Here is a straightforward example of a class `point`.

```
let point = class (x0)
  val x = ref x0
  meth move d = (x := !x + d; !x)
end;;
```

value point : class (int)
  val x : int ref
  meth move : int → int
end

Class types are automatically inferred. Objects are usually created as instances of classes. All objects of the same class have the same type structure, reflecting the structure of the class. It is important to name object types to avoid repeating the whole nested, often recursive, structure of objects at each occurrence of an object type. Thus, the above declaration also automatically defines the abbreviation:

```
type point = ⟨move : int → int⟩
```

which is the type of objects with a method `move` of type `int → int`. In practice, this is essential in order to report readable types to the user.

```
let p = new point 3;;
```

value p : point = ⟨obj⟩

Classes can also be derived from other classes by adding instance variables and methods. The following example also shows how an object sends messages to itself; for instance, if the `scale` formula is overridden in a subclass, the `move` method will use the new scale. Here, methods of the parent class are bound in the super class variable `parent` and used in the redefinition of the `move` method.

```
let scaled_point = class (s0)
  inherit point 0 as parent
  val  s = s0
  meth scale = s
  meth move d =
    parent#move (d * self#scale)
end;;
```

value scaled_point : class (int)
  val s : int
  val x : int ref
  meth move : int → int
  meth scale : int
end

Scaled points have a richer interface than points. It is still possible to consider scaled points as points. This might be useful if one wants to mix different kinds of points with incompatible attributes, ignoring anything but the interface of points (we assume that a class of colored points has been defined in an obvious way):

```
let points =
  [(new scaled_point 2 : scaled_point :> point);
   (new colored_point "blue"
    : colored_point :> point)];;
```

value points : point list = [⟨obj⟩; ⟨obj⟩]

Notation

A binding is a pair (k, t) of a key k and an element t. It is written $k = t$ when t is a term or $k : t$ when t is a type. Bindings may also be tagged. For instance, if `foo` is a tag, we write `foo` $u = a$ or `foo` $u : a$. Tags are always redundant in bindings and only used to remind which kind of identifier is bound.

Term sequences may contain several bindings of the same key; we write @ the concatenation of sequences (i.e. their juxtaposition). On the contrary, linear sequences cannot bind the same key several times. We write $+$ the overriding extension of a sequence with another one, and $\oplus$ to enforce that the two sequences must be compatible (i.e. they must agree on the intersection of their domains). We write $\emptyset$ for the empty sequence.

A sequence can be used as a function. More precisely, the domain of a sequence S is the union, written $dom(S)$, of the first projection of the elements of the sequence. An element of the domain k is mapped to the value t so that $x : t$ is the rightmost element of the sequence whose first projection is x, ignoring the tags. The sequence $S \setminus$ `foo` is composed of all elements of S but those tagged with `foo`. Finally, we write `foo` (S) for $\{k : t \mid \texttt{foo}\ k : \tau \in S\}$, the subsequence of the elements of S tagged with `foo` but stripped of the tag `foo`.

We write $\bar{t}$ for a tuple of elements $({t_i}^{i \in I})$ when indexes are implicit from the context.

2 Objects

We assume that a set of variables $x \in \mathcal{X}$ and two sets of names $u \in \mathcal{U}$ and $m \in \mathcal{M}$ are given. Variable x is used to abstract other expressions; x is bound in `fun` $(x)\ a$ and `let` $x = a_1$ `in` a_2. Programs are considered equal modulo renaming of bound variables. Conversely names are always free and not subject to α-conversion: u and m are used to name value and method fields of objects, respectively. The syntax of expressions is provided below.

$$
\begin{array}{l}
a ::= x \mid \texttt{fun}\,(x)\; a \mid a\; a \mid \texttt{let}\; x = a\; \texttt{in}\; a \\
\quad \mid \texttt{self} \mid u \mid \{\langle u = a; \ldots u = a\rangle\} \mid a\#m \\
\quad \mid \langle \texttt{val}\; u = a\,; \ldots \texttt{val}\; u = a; \\
\qquad \texttt{meth}\; m = a\,; \ldots \texttt{meth}\; m = a\rangle
\end{array}
$$

Operations on references could be included as constants k (the ellipsis in syntax definitions means that we are extending the previous definition):

$$a ::= \ldots \mid k \qquad \text{and} \qquad k ::= \texttt{ref}\;\_ \mid (\_ := \_) \mid (!\_)$$

For sake of simplicity, we omit them in the formalization, although we use them in the examples. An object is composed of a hidden internal state which in turn is composed of a sequence of *value* bindings also called *instance variables*, and a sequence of *method* bindings for accessing and modifying these values. The type of an object is thus the type of the record of its methods. In an object, a method may return the object itself or expect to be applied to another object of the same kind. Types might thus be recursive. We assume given two countable collections of type variables and row variables, written α and ρ, and a collection of type constructors written κ.

$$
\begin{array}{l}
\tau ::= \alpha \mid \tau \to \tau \mid (\tau, \ldots \tau)\; \kappa \mid \texttt{rec}\; \alpha.\tau \mid \langle \tau' \rangle \\
\tau' ::= (m : \tau; \tau') \mid \rho \mid \emptyset \\
\sigma ::= \forall\, \bar{\alpha}.\, \tau
\end{array}
$$

Object types ending with a row variable are named *open object types*, while others are named *closed object types*. In the examples, closed object types are simply written $\langle m_i : {\tau_i}^{i \in I}\rangle$, i.e. the $\emptyset$ symbol is omitted, and the row variables of open object types are also left implicit in an ellipsis $\langle m_i : {\tau_i}^{i \in I}; ..\rangle$ (abbreviations explained in section 7 can even be used to share ellipsis). In the formal presentation, we keep both $\emptyset$ and row variables explicit. A label can only appear once in an object type, which is easily guaranteed by sorting type expressions [Rém93].

Type equality is defined by the following family of left-commutativity axioms:

$$(m_1 : \tau_1; m_2 : \tau_2; \tau') = (m_2 : \tau_2; m_1 : \tau_1; \tau')$$

plus standard rules for recursive types [AC93]. Types, sorts, and type equality are a simplification of those used in [Rém94a], which we refer to for details.

Typing contexts are sequences of bindings:

$$A ::= \emptyset \mid A + x : \sigma \mid A + \texttt{val}\; u : \tau$$

Typing judgments are of the form $A \vdash a : \tau$. The typing rules for ML are recalled in appendix A.

Typing rules for objects are given in figure 1. A simple object is just a set of methods. Methods can send messages to the object itself, which will be bound to the special variable `self`. A simple object could be typed as follows:

$$\frac{A + \texttt{self} : \langle m_j : {\tau_j}^{j \in J}\rangle \vdash a_j : {\tau_j}^{j \in J}}{A \vdash \langle \texttt{meth}\; m_j = {a_j}^{j \in J}\rangle : \langle m_j : {\tau_j}^{j \in J}\rangle}$$

However, an object can also have instance variables. Instance variables may only be used inside methods defined in the same object. The typechecking of instance variables $\texttt{val}\; u_i : {a_i}^{i \in I}$ of an object produces a small typing environment $\texttt{val}\; u_i : {\tau_i}^{i \in I}$ in which the methods are typed (rules OBJECT and VAL).

Instance variables also provide the ability to clone an object possibly overriding some of its instance variables (rule OVERRIDE). In this rule, types τ_y and τ_i do not seem to be connected, but in fact they are, thanks to typing rule OBJECT which requires the type τ_y of `self` and the types τ_i of instance variables to be related to the same objet. This is also insured by typing the premise in the context $A^\star$ equal to $A \setminus \{\texttt{val}, \texttt{self}\}$. This makes the expression $\langle \texttt{val}\; u = a\,; \texttt{meth}\; m = \langle \texttt{meth}\; m = u\rangle\rangle$ ill-typed; but is not a real restriction, since one can still write the less ambiguous expression $\langle \texttt{val}\; u = a\,; \texttt{meth}\; m = \texttt{let}\; x = u\; \texttt{in}\; \langle \texttt{meth}\; m = x\rangle\rangle$.

The rule SEND for message invocation is similar to the rule for polymorphic access in records: when sending message m to an object a, the type of a must be an object type with method m of type τ; the object may have other methods that are captured in the row expression τ'. The type returned by the invocation of the message is τ. The type of message invocation may also be seen on a trivial program:

```
let send_m a = a#m;;
```
value send_m : ⟨ m : 'a; .. ⟩ → 'a = ⟨fun⟩

The ellipsis stands for an anonymous row variable ρ, which means that any other method than m may also be defined in the object a. Row variables provide parametric polymorphism for method invocation. Instead, many other languages use subtyping polymorphism. Since our subtyping polymorphism must be explicitly declared (see section 4), row variables are essential here to keep type inference. Row variables also allow to express some kind of matching [Bru95] without F-bounded or higher-order quantification [PT94, AC95]. Here is an example:

```
let min x y = if x#leq y then x else y;;
```
value min :
(⟨ leq : 'a → bool; .. ⟩ as 'a) → 'a → 'a =

$$\frac{\texttt{val}\ u : \tau \in A}{A \vdash u : \tau}\ (\text{Val}) \qquad \frac{\texttt{val}\ u_i : \tau_i \in A \qquad \texttt{self} : \tau_y \in A \qquad A \vdash a_i : \tau_i}{A \vdash \{\langle u_i : a_i{}^{i \in I} \rangle\} : \tau_y}\ (\text{Override})$$

$$\frac{A^\star \vdash a_i : \tau_i{}^{i\in I} \qquad A^\star + \texttt{self} : \langle m_j : \tau_j{}^{j \in J}\rangle + \texttt{val}\ u_i : \tau_i{}^{i\in I} \vdash a_j : \tau_j{}^{j\in J}}{A \vdash \langle \texttt{val}\ u_i = a_i{}^{i\in I}\,; \texttt{meth}\ m_j = a_j{}^{j\in J}\rangle : \langle m_j : \tau_j{}^{j\in J}\rangle}\ (\text{Object}) \qquad \frac{A \vdash a : \langle m : \tau; \tau' \rangle}{A \vdash a\#m : \tau}\ (\text{Send})$$

Figure 1: Typing rules for objects

$\langle fun \rangle$

The binder `as` makes it possible to deal with open object types occurring several times in a type (this will be detailed in section 7). An expanded version of this type is:

$\texttt{rec}\ \alpha.\langle leq : \alpha \to bool; \rho\rangle \to$
$\texttt{rec}\ \alpha.\langle leq : \alpha \to bool; \rho\rangle \to \texttt{rec}\ \alpha.\langle leq : \alpha \to bool; \rho\rangle$

The function `min` can be used for any object of type τ with a method $leq : \tau \to bool$, since the row variable ρ can always be instantiated to the remaining methods of type τ.

3 Classes

The syntax for classes, introduced in section 1, is formally given in figure 2. The body of a class is a sequence b of small definitions d. We assume given a collection of class identifiers $z \in \mathcal{Z}$, and a class of super class identifiers written s.

We have also enriched the syntax of objects so that it reflects the syntax of classes. That is, objects can also be built using inheritance, and `val`-fields need not precede `meth`-fields.

In practice, classes will only appear at toplevel. However, it is simpler to leave more freedom. For instance, they may appear under abstraction. Classes are not first order though, since they can be bound but not abstracted. Technically, classes could be first order, and abstracted over with explicit polymorphism. However, their compilation would become more difficult, or less efficient, and the gain is probably not worth the complication.

The type of a class, $\texttt{class}\ (\tau) : \tau_y\ \langle\varphi\rangle$, is composed of the type τ of its initialization argument, the type τ_y of self (i.e. the type an object of this class would have), and the type φ of its field bindings and method bindings. Class types are written γ. Type schemes are extended with class types.

$\gamma ::= \texttt{class}\ (\tau) : \tau\ \langle\varphi\rangle$
$\varphi ::= \emptyset \mid \varphi\,; \texttt{val}\ u : \tau \mid \varphi\,; \texttt{meth}\ m : \tau \mid \varphi\,; \texttt{super}\ s : \varphi$
$\sigma ::= \ \ldots \mid \forall\,\bar\alpha.\,\gamma$

Typing contexts are extended with class variables bindings and super-class bindings: :

$$A ::= \ \ldots \mid A + z : \sigma \mid A + \texttt{super}\ s : \varphi$$

We add new typing judgments $A \vdash b : \varphi$ and $A \vdash d : \varphi$ that are used to type class bodies. We also redefine $A^\star$ to be A where all `val`, `meth`, `super` and `self` bindings have been removed. Typing rules are given in figure 3. Generalization $\texttt{Gen}(\gamma, A)$ is, as in ML, $\forall\,\bar\alpha.\,\gamma$ where $\bar\alpha$ are all variables of γ that are not free in A.

Classes are typed by adding items (inheritance clauses, value or method fields) one after the other. Value fields are typed in $A^\star$, so that they cannot depend on other value fields (rule Value). On the opposite, methods may depend on all previously defined instance variables and super classes (rule Method). The Inherit rule ensures that `self` is assigned the same type in both the superclass and the subclass; all bindings of the superclass are discharged in the subclass, and the super variable is bound to the types of the fields of the superclass. Super variables are only visible while typechecking the body of the class but are not exported in the type of the class itself, as shown by rule Then. The rule Object is more general than (and overrides) the one of figure 1; it corresponds to the combination of rule Class and rule New.

When a value or method field is redefined, its type cannot be changed, since previously defined methods might have assumed the old type. This is enforced by using in rule Then the $\oplus$ operator which requires that the two argument sequences be compatible on the intersection of their domains. At first, this looks fairly restrictive. But it still leaves enough freedom in practice. Indeed, the class type can also be specialized by instantiating some type variables.

$$
\begin{array}{ll}
a ::= \ \ldots \mid \langle b \rangle \mid \texttt{let}\ z = c\ \texttt{in}\ a \mid \texttt{new}\ c \mid s\#m & \text{Expressions} \\
c ::= z \mid \texttt{class}\ (x)\ b & \text{Class expressions} \\
b ::= \emptyset \mid d\,;b & \text{Class bodies} \\
d ::= \texttt{inherit}\ c\ a\ \texttt{as}\ x\ \texttt{as}\ s \mid \texttt{val}\ u = a \mid \texttt{meth}\ m = a &
\end{array}
$$

Figure 2: Core class syntax

(Value)
$$\frac{A^\star \vdash a : \tau}{A \vdash \texttt{val}\ u = a : (\texttt{val}\ u : \tau)}$$

(Method)
$$\frac{A \vdash \texttt{self} : \langle m : \tau; \tau' \rangle \qquad A \vdash a : \tau}{A \vdash \texttt{meth}\ m = a : (\texttt{meth}\ m : \tau)}$$

(Inherit)
$$\frac{A^\star \vdash a : \tau \qquad A \vdash \texttt{self} : \tau_y \qquad A \vdash c : \texttt{class}\ (\tau)\ :\ \tau_y\ \langle \varphi \rangle}{A \vdash \texttt{inherit}\ c\ a\ \texttt{as}\ s : \varphi + (\texttt{super}\ s : \varphi)}$$

(Basic)
$$\frac{}{A \vdash \emptyset : \emptyset}$$

(Then)
$$\frac{A \vdash d : \varphi_1 \qquad A + (\varphi_1 \setminus \texttt{meth}) \vdash b : \varphi_2}{A \vdash d\,; b : (\varphi_1 \setminus \texttt{super}) \oplus \varphi_2}$$

(Class)
$$\frac{A^\star + x : \tau + \texttt{self} : \tau_y \vdash b : \varphi}{A \vdash \texttt{class}\ (x)\ b : \texttt{class}\ (\tau)\ :\ \tau_y\ \langle \varphi \rangle}$$

(New)
$$\frac{A \vdash c : \texttt{class}\ (\tau)\ :\ \tau_y\ \langle \varphi \rangle \qquad \tau_y = \langle \texttt{meth}\ (\varphi) \rangle}{A \vdash \texttt{new}\ c : \tau \to \tau_y}$$

(Super)
$$\frac{\texttt{super}\ s : \varphi \in A \qquad \texttt{meth}\ m : \tau \in \varphi}{A \vdash s\#m : \tau}$$

(Object)
$$\frac{A^\star + \texttt{self} : \tau_y \vdash b : \varphi \qquad \tau_y = \langle \texttt{meth}\ (\varphi) \rangle}{A \vdash \langle b \rangle : \tau_y}$$

(Class-Let)
$$\frac{A \vdash c : \gamma \qquad A + z : \texttt{Gen}(\gamma, A) \vdash a : \tau}{A \vdash \texttt{let}\ z = c\ \texttt{in}\ a : \tau}$$

(Class-Inst)
$$\frac{z : \forall \bar{\alpha}. \gamma \in A}{A \vdash z : \gamma[\bar{\tau}/\bar{\alpha}]}$$

Figure 3: Typing rules for classes

One may imagine to relax this constraint, and accept the type of the redefined method to be a subtype of the original method. One would, however, lose a property we believe important: rule Inherits shows that the type a class gives to self is an instance of the types given to self by its ancestors; as a consequence, the type of self in a class unifies with the type of any object instance of a subclass of this class.

Methods returning objects of same type as self are thus correctly typed.

```
let duplicable = class ()
  meth copy = {⟨ ⟩}
end;;
```
value duplicable : class (unit) : 'a
  meth copy : 'a
end

In this class type, `'a` is bound to the type of self, that is objects of any subclass of this class have types that match `rec` $\alpha.\langle$`copy :` α`; ..`$\rangle$. Class `duplicable` can then be inherited, and method `copy` still have the expected type (that is, the type of self).

```
let duplicable_point = class (x)
  inherit duplicable () inherit point x
end;;
```
value duplicable_point : class (int) : 'a
  val x : int ref
  meth copy : 'a
  meth move : int → int
end

Note that ancestors are ordered, which disambiguates possible method redefinitions: the final method body is the one inherited from the ancestor appearing last.

Rule Class-Let and Class-Inst are similar to the rules Let and Inst for core ML (described in appendix A). These rules are essential since polymorphism of class types enables method specialization during inheritance, as explained above.

4 Coercion

Polymorphism on row variables enables to write a parametric function that sends a message m to any object that has a method m. Thus subtyping polymorphism is not required here. This is important since subtyping cannot be inferred in Objective ML.

There is still a notion of explicit subtyping, that allows explicit coercion of an expression of type τ_1 to an expression of type τ_2 whenever τ_1 is a subtype of τ_2. As shown in the last example of section 1, this enables to see all kinds of points just as simple points, and put them in the same data-structure.

The language of expressions is extended with the following construct:

$$a ::= \ \dots \mid (a : \tau :> \tau)$$

The corresponding typing rule is:

$$\text{(Coerce)}\ \frac{\tau \leq \tau' \qquad A \vdash a : \theta(\tau)}{A \vdash (a : \tau :> \tau') : \theta(\tau')} \qquad \theta \text{ substitution}$$

The premise $\tau \leq \tau'$ means that τ is a subtype of τ'. This subtyping relation $\leq$ is standard [AC93]. We choose the simpler and more efficient presentation of Palsberg [KPS93]. The constraint $\tau \leq \tau'$ is defined on regular trees as the smallest transitive relation that obey the following rules:

Closure rules

$$\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' \Longrightarrow \tau_1' \leq \tau_1 \wedge \tau_2 \leq \tau_2'$$
$$\langle m : \tau ; \tau \rangle \leq \langle m : \tau' ; \tau_0' \rangle \Longrightarrow \tau \leq \tau'$$

Consistency rules

$$\tau \leq \tau_1 \to \tau_2 \Longrightarrow \tau \text{ is of the shape } \tau_1' \to \tau_2'$$
$$\tau \leq \langle m_i : {\tau_i}^{i \in I} \rangle \Longrightarrow \tau \text{ of the shape } \langle m_i : {\tau_i'}^{i \in I} ; \tau' \rangle$$
$$\tau \leq \langle m_i : {\tau_i}^{i \in I} p ; \rho \rangle \Longrightarrow \tau = \langle m_i : {\tau_i}^{i \in I} p ; \rho \rangle$$
$$\tau \leq \alpha \Longrightarrow \tau = \alpha$$

Our subtyping relation does not enounce typing assumptions on variables, and it is thus weaker than the subtyping relation used in [EST95b], except on ground types.

5 Semantics

We give a small step reduction semantics for our language. Values are functions or object values. Object values are composed of methods or instance variables which are themselves values; instance variables must precede methods and neither can be overridden in values. Values, evaluation contexts, and reduction rules are given in figure 4.

We have chosen to reduce inheritance in objects rather than classes. It would also be possible to reduce inheritance inside classes, and rearranged methods fields and value fields as well. Our choice is simpler and more general, since classes can also be inherited in objects. The reduction of object expressions to values is done in two passes: inheritance and evaluation of value fields are reduced top down, then fields are reordered and redundant fields are removed bottom-up. The invocation of a message evaluates the corresponding expression after replacing instance variables, self, and overriding by their current values.

Coercion behaves as an identity function: the coercion of a value reduces to the value itself. Subject reduction can then only be proved by extending the type system with an implicit subtyping rule:

$$\frac{A \vdash a : \tau \qquad \tau \leq \tau'}{A \vdash a : \tau'} \ (\textsc{Sub})$$

The soundness of the language is stated by the two following theorems.

Theorem 1 (Subject Reduction)
Reduction preserves typings (i.e. for any A such that $dom(A) \subset \mathcal{X} \cup \mathcal{Z}$, if $A \vdash a : \tau$ and $a \longrightarrow a'$ then $A \vdash a' : \tau$.)

The proof is done by mutual induction on the size of a and the size of b in the corresponding property for class bodies, i.e. if $A \vdash b : \varphi$ and $b \longrightarrow b'$ then $A \vdash b' : \varphi$. Since hiding has been done by limiting the scope of variables, there is no particular difficulties in the proof; its structure is similar to the one for core ML. Provided a few substitution lemmas, all cases for reducing class bodies are immediate. The instantiation of a class is also an easy case, since the rule for typing objects is derived from the rule for typing classes.

Theorem 2 (Normal forms) *Well-typed irreducible normal forms are values* (i.e. if $\emptyset \vdash a : \tau$ and a cannot be reduced, then a is a value.)

6 Type inference

The syntax forbids class abstraction. Indeed, in an expression `fun` (x) a, variable x will never be bound to a class. This ensures that class types never need to be guessed. Polymorphism is only introduced at Let binding of classes or values. Thus, type inference reduces to first-order unification on types, as in ML. Types of Objective ML are a restriction of record types. First-order unification for record types is decidable, and solvable unification problems admits principal solutions, even in the presence of recursion [Rém94a]. Consequently, Objective ML has the principal type property.

Values

$$v ::= \ldots \mid \texttt{fun}\,(x)\ a \mid \langle w\rangle \mid \texttt{class}\,(x)\ b$$
$$w ::= \emptyset \mid w_d\ ;w \qquad\qquad \texttt{val}\text{ fields preceed }\texttt{meth}\text{ fields, no overridings}$$
$$w_d ::= \texttt{meth}\ m = a \mid \texttt{val}\ u = v$$

Evaluation contexts

$$E ::= [] \mid \texttt{let}\ z = E\ \texttt{in}\ d \mid E\ a \mid v\ E \mid E\#m \mid \langle F\rangle \mid \texttt{new}\ E$$
$$F ::= [] \mid F_d\ ;b \mid w_d\ ;F$$
$$F_d ::= \texttt{inherit}\ c\ E\ \texttt{as}\ s \mid \texttt{val}\ u = E$$

From classes to objects

$$\texttt{new}\,(\texttt{class}\,(x)\ b) \longrightarrow \texttt{fun}\,(x)\ \langle b\rangle$$

Reduction of objects

$$\texttt{inherit}\,(\texttt{class}\,(x)\ b)\ v\ \texttt{as}\ s\ ;b' \longrightarrow b[v/x]\ @\ (b'\,[b(m)[v/x]/s\#m]^{m\in dom\,(b)})$$
$$\texttt{val}\ u = v\ ;w \longrightarrow w \qquad \text{if } u \in dom\,(w)$$
$$\texttt{meth}\ m = a\ ;w \longrightarrow w \qquad \text{if } m \in dom\,(w)$$
$$\texttt{meth}\ m = a\ ;(\texttt{val}\ u = v\ ;w) \longrightarrow \texttt{val}\ u = v\ ;(\texttt{meth}\ m = a\ ;w)$$

Reduction of message invocation ($U = dom\,(w)$)

$$\langle w\rangle\#m \longrightarrow w(m)[\langle w\rangle/\texttt{self}][w(u)/u]^{u\in U}[\langle w\ @\ (\texttt{val}\ u = {a_u}^{u\in V})\rangle/\{\langle u = {a_u}^{u\in V}\rangle\}]^{V\subset U}$$

Reduction of coercions

$$(v : \tau :> \tau') \longrightarrow v$$

Reduction of other expressions

$$\texttt{let}\ x = v\ \texttt{in}\ a \longrightarrow a[v/x] \qquad \texttt{let}\ z = v\ \texttt{in}\ a \longrightarrow a[v/z]$$
$$(\texttt{fun}\,(x)\ a)\ v \longrightarrow a[v/x]$$

Context reduction

$$E[a] \longrightarrow E[a'] \text{ if } a \longrightarrow a' \qquad F[b] \longrightarrow F[b'] \text{ if } b \longrightarrow b'$$

Figure 4: Semantics

Theorem 3 (Principal types) *For any typing context A and any program a that is typable in the context A, there exists a type τ such that $A \vdash a : \tau$ and for any other type τ' such that $A \vdash a : \tau'$ there exists a substitution θ whose domain does not intersect the free variables of A and such that $\tau' = \theta(\tau)$.*

7 Abbreviations

Object types tend to be very large. Indeed, the type of an object lists all its method types, which can themselves contain other object types. This quickly becomes unmanageable [Rém94a, EST95a]. Introducing abbreviations is thus of crucial importance. However, usual ML type abbreviations are not powerful enough: they are expanded and lost during unification and they cannot be recursive. We have thus enhanced them.

We view abbreviations as name aliases for types. In order to trace abbreviations, we consider types as graphs where abbreviations are names for shared nodes. In addition to the usual notation $\texttt{rec}\ \alpha.\tau$, the construct $(\tau\ \texttt{as}\ \alpha)$ is also used to bind α to τ. Since α is also bound outside of τ, this is a notation for graphs, and not only for regular trees. That is, the two types $(\langle m : \alpha\rangle\ \texttt{as}\ \alpha') \rightarrow \alpha'$ and $(\langle m : \alpha\rangle) \rightarrow (\langle m : \alpha\rangle)$ are different graphs, that represent the same regular tree. Such a finer representation of types is necessary in order to keep types small and abbreviated.

Types aliases can also be used to simplify object types. Rather than sorting types to ensure they are well-formed, a stronger condition is to require two object types that share the same row variable to be equal. This rules out incorrect types such as $\langle \rho\rangle \rightarrow \langle m : \tau; \rho\rangle$. Types such as $\langle m : \tau_1; \rho\rangle \rightarrow \langle m : \tau_2; \rho\rangle$, at the basis of record extension, are also rejected. However, there is no primitive operation on objects that would have such a type. These types can thus be ruled out without serious restriction of the language. Moreover all programs still keep the same principal types.

This restriction has been chosen in the implementation since it avoids explaining sorts to the user. It also makes the syntax for types somewhat clearer, as row variables can then always be replaced by ellipsis. Sharing can still be described with aliasing. For instance, type $\langle m : \tau; \rho\rangle \rightarrow \langle m : \tau; \rho\rangle$ is written $(\langle m : \tau; ..\rangle\ \texttt{as}\ \alpha) \rightarrow \alpha$.

In the following example, the nodes between the argument type and the result type are shared, otherwise the ellipsis could not be used.

```
let bump x = x#move 1; x;;
```
value bump :
  $(\langle\ move : int \rightarrow\ 'b;\ ..\ \rangle\ as\ 'a) \rightarrow\ 'a = \langle fun\rangle$

Then, during the typing of the expression `bump p` below, type `(⟨move : int → 'b; ..⟩ as 'a)` and type `point` are identified. The type of `bump p` is thus also abbreviated to `point`.

```
let p = new point 7;;
```
value p : *point* = $\langle obj\rangle$
```
bump p;;
```
$-$: *point* = $\langle obj\rangle$

A class definition $\texttt{let}\ c = \texttt{class}\ (x)\ b\ \texttt{in} \ldots$ automatically generates an abbreviation for the type of its instances. For specifying it, one actually needs to add type parameters to class definitions, corresponding to the one of the abbreviation. That is, we should write

$$\texttt{let}\ (\bar{\alpha})\ c = \texttt{class}\ (x)\ b\ \texttt{in} \ldots \qquad (1)$$

where the parameters $\bar{\alpha}$ must appear in b.

In fact, abbreviations are generated from class *types*. It follows from type inference that the class definition $\texttt{class}\ (x)\ b$ has a principal class type $\texttt{class}\ (\tau_0) : \tau_y\ \langle\varphi\rangle$. Here τ_y is the type matched by objects in all subclasses. It is always of the form $\langle m_i : \tau_i{}^{i\in I}; \tau\rangle$ where $\texttt{meth}\ (\varphi)$ is a subsequence of $m_i : \tau_i{}^{i\in I}$ and τ is either $\emptyset$ (this is a pathological case, where the class cannot be extended with new methods) or a row variable ρ. If $\texttt{meth}\ (\varphi)$ is exactly $m_i : \tau_i$, then it is possible to create objects of that class; they will have type $\tau_y[\emptyset/\rho]$. Otherwise, the class is virtual and can only be inherited in other class definitions. If all free type variables of τ_y except ρ are listed in $\bar{\alpha}$, we automatically define the following two abbreviations:

$$\texttt{type}\ (\bar{\alpha}, \rho)\ \#\kappa_c = \tau_y \qquad \texttt{type}\ (\bar{\alpha})\ \kappa_c = (\bar{\alpha}, \emptyset)\ \#\kappa_c$$

The former matches all objects of subclasses of c. The latter is a special case of the former, and abbreviates any objects of class c.

Let us consider an example. Class `point` has type $\texttt{class}\ (\tau_0) : \tau_y\ \langle\varphi\rangle$ with $\tau_y = \langle \texttt{move} : \texttt{int} \rightarrow \texttt{int}; \rho\rangle$, for some τ_0 and φ. The two following abbreviations have thus been defined for this class:

$$\texttt{type}\ \rho\ \#\texttt{point} = \langle \texttt{move} : \texttt{int} \rightarrow \texttt{int}; \rho\rangle$$

$$\texttt{type}\ \texttt{point} = \langle \texttt{move} : \texttt{int} \rightarrow \texttt{int}\rangle$$

One can check that type `point` is an abbreviation for the type of objects of class `point`, and that the type of an object of any subclass of class `point` is an instance of type $\rho\ \#\texttt{point}$.

In the concrete syntax, the row variable ρ is treated anonymously (as an ellipsis) and omitted. The former abbreviation $\#\kappa_c$ is given lower priority than regular ones in case of a clash, and vanishes as soon as the row variable is instantiated, so as to reveal the value taken by the row variable.

In fact, we allow κ_z and $\#\kappa_z$ to occur in the definition of b. To insure that abbreviations always expand to regular trees, we require that in an abbreviation definition $\texttt{type}\ (\bar{\alpha})\ \kappa = \tau$ the abbreviation κ occurs in the body τ with the same parameters $\bar{\alpha}$. This condition extends to mutually recursive abbreviations. The previous definitions can be rewritten to handle the general case correctly.

Type abbreviations are generalized to allow constraints on the type parameters of the abbreviations. This is an extension of the abbreviations of LCS [Ber93], that were also used in [Rém94a]. This is very natural as, for instance, a sorted list of comparable objects should be parameterized by the type of its elements, which is not a type variable. Moreover this extension makes it possible to avoid row variables as type parameters (as the whole object type can appear as a parameter). Here is an example:

$$\texttt{type}\ \tau_0\ \kappa = \tau$$

where all free variables of type τ appears in type τ_0. A parameter τ_1 of type constructor κ must be an instance $\theta(\tau_0)$ of type τ_0 (for some substitution θ). Then, type $\tau_1\ \kappa$ expands to type $\theta(\tau)$.

Constrained type abbreviations are also convenient since, in a class definition $\texttt{class}\ (\bar{\alpha})\ (x)\ b$, class type parameters $\bar{\alpha}$ may have been instantiated to some types $\bar{\tau}_\alpha$ while inferring the class type $\texttt{class}\ (\tau_0) : \tau_y\ \langle\varphi\rangle$. The two abbreviations generated by the class definition are thus actually:

$$\texttt{type}\ (\overline{\tau_\alpha}, \rho)\ \#\kappa_z = \tau_y \qquad \texttt{type}\ (\bar{\alpha})\ \kappa_z = (\bar{\alpha}, \emptyset)\ \#\kappa_z$$

The latter is unchanged except that the constraints of the first ones are implicit in the second one.

Class types are shown to the user stripped at their type parameters; parameters that constraint the type abbreviations are described by constraint clauses:

```
let 'a circle = class (p : 'a)
  val point = p
  meth center = point
  meth move m =
    if m = 0 then 0 else
    point#move (1 + Random.int m)
end;;
```

value circle : class 'a ('a)
  constraint 'a = ⟨ move : int → int; .. ⟩
  val point : 'a
  meth center : 'a
  meth move : int → int
end

This class defines the abbreviation

$$\texttt{type}\ (\langle \texttt{move} : \texttt{int} \to \texttt{int}; \rho\rangle\ \texttt{as}\ \alpha)\ \texttt{circle} = \langle \texttt{center} : \alpha; \texttt{move} : \texttt{int} \to \texttt{int}\rangle$$

The abbreviations inferred seem to depend on the way aliases are managed. For instance, the two types $(\tau_1\ \texttt{as}\ \alpha) * \alpha$ and $(\tau_1\ \texttt{as}\ \alpha) * \tau_1$ are equal when viewed as terms. The two alternatives lead to the two following abbreviations:

$$\texttt{type}\ (\tau_1\ \texttt{as}\ \alpha)\ \kappa = \alpha * \alpha \qquad \texttt{type}\ (\tau_1\ \texttt{as}\ \alpha)\ \kappa = \alpha * \tau_1$$

However, this does not matter as abbreviations of different views of the same type are equivalent, in the sense that they expand to equal terms.

There is a canonical way to derive best aliasing: efficient unification algorithm are usually formalized as rewriting systems over multi-sets of multi-equations. In this approach, canonical forms of a unification problem represent principal term-solutions, but also and more straightforwardly principal graph solutions. Roughly speaking, all aliasing from the input problem is kept (correctness) and the minimal aliasing is introduced in the output problem (completeness).

However, in practice only some sharing should be kept. The user can deal with sharing inside a class or a value definition, but sharing reveals too much information to the outside. In particular, core-ML programs should not use aliases. Thus, all aliasing is removed before generalization, except aliasing of open object types (so that row variables can still be printed as ellipsis) and aliasing defining recursive types.

8 Extensions

This section lists other useful features of Objective ML that have been added to the implementation. Imperative features have been ignored in the formal presentation since their addition is theoretically well-understood and independent of the presence of objects and classes. Other features are less important in theory, but still very useful in practice: private instance variables, coercion primitives.

8.1 Imperative features

We have intendedly used references in the very first example. We did not formalize references in the presentation of Objective ML, since we preferred to keep the presentation simple and focussed on objects and classes. The addition of imperative features to Objective ML is theoretically as simple as its addition to ML, and it is practically as useful.

In fact, the implementation Objective Caml also allows value fields to be mutable in a similar way mutable records were treated in Caml Special Light. For instance, we could have written:

```
let point = class (x0)
  val mutable x =  x0
  meth move d = (x ← x + d; x)
end;;
```

value point : class (int)
val mutable x : int
meth move : int → int
end

Objective Caml only allows generalization of values (actually, a slightly more general class of non expansive expressions); the creation of an object from a class `c` is not considered as a value (as it is the application of function `new c` to some arguments), so mutable fields in classes are typed as any other fields, except that mutability properties are also checked during typechecking.

8.2 Local bindings

As shown by the evaluation rules for objects, both value and method fields are bound to their rightmost definition. All value fields must still be evaluated even though they are to be discarded.

Object-oriented languages often offer more security through private instance variables. The scope of an instance variable can be restricted so that this instance variable is not visible in subclasses.

This section presents local bindings, that are only visible in the body of the class they appear in. This is weaker than what one usely expects from private instance variable, as a class cannot, for instance, inherit of an instance variable and hide it from its subclasses. Private instance variables would actually not be difficult to add. However, hiding methods in subclasses conflicts with late binding and a flat method name space.

The syntax is extended as follows:

$$\begin{aligned} d &::= \ \ldots \mid \texttt{local}\ x = a\ \texttt{in}\ b \\ F_d &::= \ \ldots \mid \texttt{local}\ x = E\ \texttt{in}\ b \end{aligned}$$

with the corresponding typing rule:

$$\frac{A^\star \vdash a : \tau \qquad A + x : \tau \vdash b : \varphi}{A \vdash \texttt{local}\ x = a\ \texttt{in}\ b : \varphi}\ (\textsc{Local})$$

Local bindings are reduced top-down, like inheritance:

$$\texttt{local}\ x = v\ \texttt{in}\ b; b' \longrightarrow b[v/x] + b'$$

In practice, however, local bindings would rather be compiled as anonymous instance variables. This would make methods independent of local bindings.

Initialization parameters could also be seen as local bindings in the whole class body, and could also be compiled as anonymous instance variables. For instance, the definition

```
let point = class (y) meth x = y end;;
```

could be automatically transformed into the equivalent program:

```
let point = class (y)
  local y = y in meth x = y
end;;
```

That way, the method `x` becomes independent of the initialization parameter `y`. Then, classes can be reduced to class values: inheritance is reduced to local bindings, local bindings are flatten, and method overriding is resolved.

8.3 Coercion primitives

Explicit coercions require both the domain and co-domain to be specified. This eliminate the need for subtype inference. In practice, however, it is often sufficient to indicate the domain of the coercion only, the co-domain of the coercion being a function S of its domain.

For convenience, we introduce a collection of coercion primitives:

$$(\_ :> \tau) : \forall\, \alpha.\, S(\tau) \to \tau$$

where α are free variables of $S(\tau)$ and τ, and $S(\tau)$ is defined as follows:

- We call positive the occurrences of a term that can be reached without traversing an arrow on the left. (This is more restrictive than the usual definition, where the arrow is treated contravariantly).
- For non recursive terms, we define $S_0(\tau)$ to be τ where every closed object type that occurs positively is opened by adding a fresh row variable.
- Terms with aliases are viewed as graphs, or equivalently as of pair of a term τ_0 and a list of constraints $\alpha_i = \tau_i$.

  Let θ be a renaming of variables α_i into fresh variables.

  Let τ'_i be τ_i in which every positive occurrence of each α_i is replaced by $\theta(\alpha_i)$.

  We return $(S_0(\tau'_0), \{\theta(\alpha_i) = S_0(\tau'_i), i \in I\} \cup \{\alpha_i = \tau_i, i \in I\})$ for $S(\tau)$.

For example,

$$S(\langle m_1 : \langle m_2 : \mathtt{int}\rangle \to \langle m_3 : \mathtt{bool}\rangle\rangle) = \langle m_1 : \langle m_2 : \mathtt{int}\rangle \to \langle m_3 : \mathtt{bool}; \rho_3\rangle; \rho_1\rangle$$

$$S(\langle m : \alpha\rangle \ \mathtt{as}\ \alpha) = \langle m : \alpha'; \rho\rangle \ \mathtt{as}\ \alpha'$$

$$S(\langle m : \alpha \to \alpha\rangle \ \mathtt{as}\ \alpha) = \langle m : (\langle m : \alpha \to \alpha\rangle \ \mathtt{as}\ \alpha) \to \alpha'; \rho\rangle \ \mathtt{as}\ \alpha'$$

The operator S has the two following properties:

$$(1) \quad S(\tau) \leq \tau \qquad (2) \quad \exists\theta\ (\theta(S(\tau)) = \tau \wedge \theta(\tau) = \tau)$$

The former gives the correctness of the reduction step $(a :> \tau) \longrightarrow (a : S(\tau) :> \tau)$. The later shows that if a has type τ then $(a :> \tau)$ also has type τ.

There is no principal solution for an operator S satisfying (1). Consider τ to be $\langle m : \mathtt{int}\rangle \to \mathtt{int}$. There are only two solutions, $\langle m : \mathtt{int}\rangle \to \mathtt{int}$ and $\langle\rangle \to \mathtt{int}$ and none is an instance of the other. This counter-example shows the weakness of the simulation of subtyping with row variables, especially on negative occurrences. There are other examples of failure on positive occurrences, but only using recursive types.

This justifies to exclude semi-explicit coercions from the core language and to treat them as a collection of primitives.

9 Comparison to other works

The closest work to Objective ML is ML-ART [Rém94a]. Here, object types are also based on record types and have the same expressiveness. State abstraction is based on explicit existential types in ML-ART; In Objective ML, it is obtained by scope hiding, but it could also be explained with a simple form of type abstraction. No coercion at all were permitted in ML-ART between objects with different interfaces. Unfortunately, ML-ART has no type-abbreviation mechanism. This was a major drawback, which motivated the design of Objective ML. On the other hand, classes are first class in ML-ART, but we do not think this is a major advantage.

Another simplification in Objective ML is that in classes all methods view self with the same type. This is not required by the semantics, and could technically be relaxed by making method types more detailed in classes (see [Rém94a]). We found that this extra flexibility is not worth the complication of class types.

Our object types are a simplification of those used in [Rém94b]. The simplification is possible since object types are similar to record types for polymorphic access, and do not require the counterpart of record extension. Moreover, as discussed above, the implementation assumes the stronger condition that two object types sharing the same row variable are always identical. With this restriction, object types seem to be equivalent to the kinded record types of Ohori [Oho90]. Ohori also proposed an efficient compilation of polymorphic records (which does not scale up to extensible records) in [Oho96]. However, his approach, based on the correspondence between types and domains of records does not scale up to implicit subtyping, and cannot be applied to the compilation of objects with code-free coercions.

Objects have been widely studied in languages with higher-order types [CCH+89, MHF93, Bru95, AC95, PT94, BM96]. Those proposals significantly differ from Objective ML. Types are not inferred but explicitly given by the user. Type abbreviations are also the user's responsibility. On the contrary, all these proposals allow for implicit subtyping.

Open record types are connected to the notion of matching introduced by Kim Bruce [Bru95, BSvG95]. Matching seems to be at least as important as subtyping in object-oriented languages. Row variables in object types express width matching in a very natural way. While explicit matching may require too much type information, type inference makes object matching very practical.

Palsberg has proposed type inference [Pal94] for a first-order version of Abadi and Cardelli's calculus of primitive objects [AC94]. However, that language is missing important features from the higher-order version [AC95]. Type inference is based on subtyping constraints and the technique is similar to the one used in [EST95a]. This later proposal [EST95a, EST95b] is closer to a real programming language, and more suited for comparison. Here, the authors use a subtyping relation that is more expressive than ours, as they can prove subtyping under some assumptions. They can also infer coercions. However, the types they infer tend to be huge. Indeed, they do not have an abbreviation mechanism. Their inheritance is weaker than ours since they must explicitly list all inherited methods in subclasses. We think the two proposals are complementary and could benefit from one another. In particular, it would be interesting to adapt automatic type abbreviations to constraint types. The problem is still nontrivial since inferred type-constraints are hard to read even in the absence of objects.

In [Dug95], Duggan proposes another approach to objects. Methods must be predeclared with a particular type scheme. Thus methods carry type information alike data-type constructors in ML. For instance, `move` would be assigned type scheme $\forall\,\alpha_y.\alpha_y \to int$. Type schemes that are assigned to methods are polymorphic in α_y: they are arrow types whose domain is always α_y, standing for the type of self. Object types only list the methods that objects of that type must accept. For instance, `point` would be given type $\langle$`move`$\rangle$. This proposal requires more type information from the user than ours. It also forbids the use of the same method name in different objects with unrelated types. Objects of parameterized classes are treated especially, using constructor kinds. Objects of a parameterized class reveal for ever that they are parameterized. For instance, let consider a class of vectors parameterized over the type α. All methods of that class must be given a type scheme of the form: $\forall\ {\alpha_\kappa}^{Type \to Type} . \forall \alpha. \alpha\ \alpha_\kappa \to \tau$, where variable α_κ range over type constructors. That is, instead of the type τ_y of `self`, only the type constructor κ_y of the type of τ_y is hidden. This reveals the dependence of τ_y on its parameters, and the parameters themselves. Methods of parameterized classes are incompatible with methods of non-parameterized classes. Objects of a vector class of characters cannot be related to objects of a string class even though they might have the same interface. In Objective ML, two such objects could be mixed. However, Objective ML, does not allow polymorphic methods while Duggan's proposal allow them. A polymorphic method `map` could be declared with type scheme: $\forall\ {\alpha_\kappa}^{Type \to Type} . \forall \alpha. \forall \alpha_1. \alpha\ \alpha_\kappa \to (\alpha \to \alpha_1) \to \alpha_1\ \alpha_\kappa$. Intuitively, `map` carries implicit universal intros and elims, like data constructors carry arguments of existentially or universally quantified types provided in [LO92, Rém94a, Läu96].

In [BM96], Bourdoncle and Metz propose a language based on some restricted form of type constraints [EST95b]. However, although their types are not higher-order, they do not provide type inference.

In Object SML [RR96b], Reppy and Riecke treat objects as a generalized form of ML concrete data types. Objects are tagged with constructors that carry the class they originated from. Therefore, objects can be tested for membership to some arbitrary class in some inheritance relationship. Only single inheritance is allowed. The subtyping relation between objects is declared and corresponds to the inheritance forest. Types are inferred, but the authors do not claim a principal type property. Some object coercions are implicit. However, all messages must give the class of the object to which messages are sent. Typing of binary methods is also a problem, which is solved via runtime class-type tests. Object SML does not provide any inheritance mechanism, except by means of encodings [RR96a].

Conclusion

Objective ML has been designed to be the core of a real programming language. Indeed, the constructs presented here have been implemented in the language Objective Caml. We chose class-based objects since this approach is now well understood in a type framework and it does not necessitate higher-order types.

The original part of the design is automatic abbreviation of object types. Although this is not difficult, it is essential for making the language practical. It has been demonstrated before that fully inferred object types are unreadable [Rém94a, EST95a]. On the contrary, types of Objective ML are clear and still require extremely little type information from the user. To our knowledge, all other existing approaches require more type declarations.

Objective ML is also interesting theoretically for the use of row variables. Row variables are very close to matching and seem more helpful than subtyping for the most common operations on objects. Message passing and inheritance are entirely based on row variables, which relegates subtyping to a lower level.

Another interesting aspect of our proposal is its simplicity. This is certainly due to the fact that Objective ML is very close to ML, in particular most features rely only on ML polymorphism. This leads to very simple typing rules for objects and inheritance. Coercions, based on subtyping, can be explained later. Data abstraction is guaranteed by scope hiding rather than by type abstraction; this is a less powerful but simpler concept.

The main drawback of Objective ML is the need for explicit coercions. Coercions are necessary. However, we think they occur in few places. Thus, explicit coercions should not be a burden. Furthermore, coercions could in theory be made implicit using constraint-based type inference.

In our implementation of Objective ML, classes and modules are fully compatible, but orthogonal. This should be particularly interesting for comparing these two styles of programming in the large, and help us to better integrate them. This is an important direction for future work.

Acknowledgment

We thank Rowan Davies who collaborated in the implementation and the design of a precursor prototype of Objective ML.

References

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *Transactions on Programming Languages and Systems. ACM*, 15(4):575–631, 1993.

[AC94] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *Theoretical Aspects of Computer Software*, pages 296–320. Springer-Verlag, April 1994.

[AC95] Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. *Science of Computer Programming*, 25(2-3):81–116, December 1995. Preliminary version appeared in D. Sanella, editor, Proceedings of European Symposium on Programming, pages 1-24. Springer-Verlag, April 1994.

[Ber93] Bernard Berthomieu. Programming with behaviors in an ML framework, the syntax and semantics of LCS. Research Report 93-133, LAAS-CNRS, 7, Avenue du Colonnel Roche, 31077 Toulouse, France, March 1993.

[BM96] Francois Bourdoncle and Stephan Merz. Primitive subtyping $\wedge$ implicit polymorphism $\models$ object-orientation. Presented at the FOOL'3 workshop, July 1996.

[Bru95] Kim B. Bruce. Typing in object-oriented languages: Achieving expressibility and safety. Revised version to appear in Computing Surveys, November 1995.

[BSvG95] Kim B. Bruce, Angela Schuett, and Robert van Gent. Polytoil: A type-safe polymorphic object-oriented language. In *ECOOP*, number 952 in LNCS, pages 27–51. Springer Verlag, 1995.

[CCH+89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[Dug95] Dominic Duggan. Polymorphic methods with self types for ML-like languages. Technical report CS-95-03,, University of Waterloo, 1995.

[EST95a] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *OOPSLA*, 1995.

[EST95b] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics*, 1995.

[KPS93] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. In *Proc. 20th symp. Principles of Programming Languages*, pages 419–428. ACM press, 1993.

[Läu96] Konstantin Läufer. Putting type annotations to work. In *Proceedings of the 23th ACM Conference on Principles of Programming Languages*, January 1996.

[Ler96] Xavier Leroy. The Objective Caml system. Software and documentation available on the Web, `http://pauillac.inria.fr/ocaml/`, 1996.

[LO92] Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, 1992.

[MHF93] John C. Mitchell, Furio Honsell, and Kathleen Fisher. A lambda calculus of objects and method specialization. In *1993 IEEE Symposium on Logic in Computer Science*, June 1993.

[Oho90] Atsushi Ohori. Extending ML polymorphism to record structure. Technical Report CSC 90/R24, University of Glasgow, Department of Computer Science, September 1990.

[Oho96] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1996.

[Pal94] Jens Palsberg. Efficient type inference of object types. In *Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 186–195, Paris, France, July 1994. IEEE Computer Society Press. To appear in Information and Computation.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal* *of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[Rém92] Didier Rémy. Extending ML type system with a sorted equational theory. Technical Report 1766, INRIA-Rocquencourt, BP 105, F-78 153 Le Chesnay Cedex, 1992.

[Rém93] Didier Rémy. Syntactic theories and the algebra of record terms. Research Report 1869, Institut National de Recherche en Informatique et Automatisme, BP 105, F-78 153 Le Chesnay Cedex, 1993.

[Rém94a] Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 321–346. Springer-Verlag, April 1994.

[Rém94b] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1994.

[RR96a] John H. Reppy and Jon G. Riecke. Classes in Object ML. Presented at the FOOL'3 workshop, July 1996.

[RR96b] John H. Reppy and Jon G. Riecke. Simple objects for Standard ML. In *Programming Language Design and Implementation 1996.* ACM, may 1996.

A Typing rules for core ML

(Inst)
$$\frac{x : \forall \bar{\alpha}. \tau \in A}{A \vdash x : \tau[\bar{\tau}/\bar{\alpha}]}$$

(Fun)
$$\frac{A + x : \tau \vdash a : \tau'}{A \vdash \texttt{fun}\ (x)\ a : \tau \to \tau'}$$

(App)
$$\frac{A \vdash a : \tau' \to \tau \qquad A \vdash a' : \tau'}{A \vdash a\ a' : \tau}$$

(Let)
$$\frac{A \vdash a' : \tau' \qquad A + x : \texttt{Gen}(\tau', A) \vdash a : \tau}{A \vdash \texttt{let}\ x = a'\ \texttt{in}\ a : \tau}$$

Generalization $\texttt{Gen}(\tau, A)$ is $\forall \bar{\alpha}. \tau$ where $\bar{\alpha}$ are all variables of τ that are not free in A.