Objective ML:
An effective object-oriented extension to ML

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

Objective ML is a small practical extension to ML with objects and top level 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 to ML with class-based objects. *Objective ML* is a fully conservative extension to 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 and let-binding polymorphism. Types are extended with object types that are similar to record types for polymorphic access. Both the status and the treatment of type abbreviations have been improved 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 that provides most features of object-oriented programming. This includes methods returning self and binary methods, of course, but also abstract classes and multiple inheritance. Coercion from objects to their counterparts in super classes is also possible. However, they must be explicit.

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 style as well as in functional style and it allows for efficient memory management (methods can be shared between all the instances of a class).

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. 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 sections 7 and 8. Extensions to the core language are presented in sections 9 and 10. In section 11, we compare our proposal with other work.

1. An overview of Objective ML

Objective ML is a core langage. An extended language based on Objective ML has been implemented on top of the Caml Special Light system [19]. This implementation is called *Objective Caml*. In this article, we completely formalize the core language, *i.e.* Objective ML. We also use the name Objective Caml to refer to the implementation, especially when describing minor differences or extension to the core language that have not been fully formalized. All examples show below have been process by Objective Caml[1]. When useful, we display the output of the typechecker in a slanted font. Toplevel definitions are implicit `let .. in ...` For each phrase, the typechecker outputs the binding that will be generalized and added to the global environment before starting to typecheck 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`.

```
class point x0 = struct
  field x = ref x0
```

```
  method move d = (x := !x + d; !x)
end;;
class point : int → sig
  field x : int ref
  method 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. The following example shows that these object abbreviations are introduced when the operator `new` is applied to a class.

```
new point;;
− : int → point = ⟨fun⟩
let p = new point 3;;
value p : point = ⟨obj⟩
```

Classes can also be derived from other classes by adding fields and methods. The following example 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 by the super-class variable `parent` and are used in the redefinition of the `move` method (the binary operator `#` denotes method invocation in Objective ML).

```
class scaled_point s0 = struct
  inherit point 0 as parent
  field  s = s0
  method scale = s
  method move d =
    parent#move (d * self#scale)
end;;
class scaled_point : int → sig
  field s : int
  field x : int ref
  method move : int → int
  method 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:

```
let points =
 [(new scaled_point 2 : scaled_point ⟨: point);
   new point 1];;
value points : point list = [⟨obj⟩; ⟨obj⟩]
```

A few other examples are given in the paper, and an example using binary methods can be found in the appendix 3.

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 are only used to remind what kind of identifier is bound.

Term sequences may contain several bindings of the same key. We write @ for the concatenation of sequences (i.e. their juxtaposition). On the contrary, linear sequences cannot bind the same key several times. We write $+$ for 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 $S \setminus \texttt{foo}$ is composed of all elements of S but those tagged with `foo`. Finally, we write `foo` (S) for $\{k : t \mid \texttt{foo}\ k : t \in S\}$, that is, for 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. Variables are used to abstract other expressions; x is bound in $\texttt{fun}\ (x)\ a$ and $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$. Programs are considered equal modulo renaming of bound variables. Names u and m are used to name field and method components of objects, respectively. Field names and method names are always free and not subject to α-conversion. The syntax of expressions is provided below.

$$
\begin{aligned}
a ::= {}& x \mid \texttt{fun}\ (x)\ a \mid a\ a \mid \texttt{let}\ x = a\ \texttt{in}\ a \\
& \mid \texttt{self} \mid u \mid \{\langle u = a; \ldots u = a\rangle\} \mid a\#m \\
& \mid \langle \texttt{field}\ u = a\,; \ldots \texttt{field}\ u = a; \\
& \qquad \texttt{method}\ m = a\,; \ldots \texttt{method}\ m = a\rangle
\end{aligned}
$$

Operations on references could be included as constants k (the ellipsis in syntax definitions means that we are extending the previous definition; "_" marks the positions of arguments around prefix or infix constants):

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

For the sake of simplicity, we omit them in the formalization, although they are used in the examples. An object is composed of a sequence of *field* bindings—the hidden internal state—, and a sequence of *method* bindings for accessing and modifying these fields. Fields are also called *instance variables*. 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.

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

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. 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 8 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. This is easily ensured by sorting type expressions [30]. The distinction between τ and $\tilde{\tau}$ can also be guaranteed by sorts. Thus, we omit the distinction and simply write τ below.

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 [4]:

(REC)
$$\frac{\tau_1 = \tau_2}{\mathtt{rec}\ \alpha.\tau_1 = \mathtt{rec}\ \alpha.\tau_2}$$

(FOLD-UNFOLD)
$$\mathtt{rec}\ \alpha.\tau = \tau[\mathtt{rec}\ \alpha.\tau/\alpha]$$

(CONTRACT)
$$\frac{\tau_1 = \tau[\tau_1/\alpha] \wedge \tau_2 = \tau[\tau_2/\alpha] \qquad \mathtt{rec}\ \alpha.\tau \text{ well-formed}}{\tau_1 = \tau_2}$$

Recursive types $\mathtt{rec}\ \alpha.\tau$ are only well-formed if τ is neither a variable nor of the form $\mathtt{rec}\ \alpha'.\tau'$ (this is not too restrictive since $\mathtt{rec}\ \alpha.(\mathtt{rec}\ \alpha'.\tau')$ can always be rewritten $\mathtt{rec}\ \alpha.\tau'[\alpha/\alpha']$). This guarantees that τ is contractive in α, and ensures that $\mathtt{rec}\ \alpha.\tau$ effectively defines a regular tree. Types, sorts, and type equality are a simplification of those used in [31], which we refer to for details. Typing contexts are sequences of bindings:

$$A ::= \emptyset \mid A + x : \sigma \mid A + \mathtt{field}\ u : \tau \mid A + \mathtt{self} : \tau$$

Typing judgments are of the form $A \vdash a : \tau$. The typing rules for ML are recalled in appendix 1.

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:

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

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 $(\mathtt{field}\ u_i = a_i)^{i \in I}$ of an object produces a typing environment $(\mathtt{field}\ u_i : \tau_i)^{i \in I}$ in which the methods are typed (rules OBJECT and FIELD).

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. They are however, 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 object. This is also ensured by typing the premises in the context $A^\star$ equal to $A \setminus \{\mathtt{field}, \mathtt{self}\}$. As a result, the expression $\langle \mathtt{field}\ u = a\,; \mathtt{method}\ m = \langle \mathtt{method}\ m = u \rangle \rangle$ is ill-typed. This is not a real restriction however, since one can still write the less ambiguous expression $\langle \mathtt{field}\ u = a\,; \mathtt{method}\ m = \mathtt{let}\ x = u\ \mathtt{in}\ \langle \mathtt{method}\ m = x \rangle \rangle$.

The rule SEND for method invocation is similar to the rule for polymorphic access in records: when sending a 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 method is τ. The type of method invocation may also be seen below:

```
let send_m a = a#m;;
value send_m : ⟨ m : α; .. ⟩ → α = ⟨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 of using row variables, many other languages use subtyping polymorphism. Since subtyping polymorphism must be explicitly declared in Objective ML (see section 4), row variables are essential here to keep type inference. Row variables also allow to express some kind of matching [7] without F-bounded or higher-order quantification [28, 2, 3]. Here is an example:

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

The binder "as" makes it possible to deal with open object types occurring several times in a type (this will be detailed

$$\frac{\text{field}\,u : \tau \in A}{A \vdash u : \tau}\ (\text{FIELD})$$

$$\frac{(\text{field}\,u_i : \tau_i \in A \qquad A \vdash a_i : \tau_i)^{i \in I} \qquad \text{self} : \tau_y \in A}{A \vdash \{\langle u_i : a_i{}^{i\in I}\rangle\} : \tau_y}\ (\text{OVERRIDE})$$

$$\frac{A^\star \vdash a_i : \tau_i{}^{i\in I} \qquad A^\star + \text{self} : \langle m_j : \tau_j{}^{j\in J}\rangle + \text{field}\,u_i : \tau_i{}^{i\in I} \vdash a_j : \tau_j{}^{j\in J}}{A \vdash \langle \text{field}\,u_i = a_i{}^{i\in I}\,;\text{method}\,m_j = a_j{}^{j\in J}\rangle : \langle m_j : \tau_j{}^{j\in J}\rangle}\ (\text{OBJECT})$$

(This rule will be overridden by the more general rule of same name in figure 3.)

$$\frac{A \vdash a : \langle m : \tau ; \tau' \rangle}{A \vdash a\#m : \tau}\ (\text{SEND})$$

FIG. 1. Typing rules for objects

in section 8). 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 as given a collection of class identifiers $z \in \mathcal{Z}$, and a collection 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 fields need not precede methods.

In practice, classes will only appear at the top level. However, it is simpler to leave more freedom, and let them appear anywhere except under abstraction. Technically, it would be possible to make them first-class, that is to allow abstraction of classes; however, class types should be provided explicitly in abstractions. The little gain in practice is probably not worth the complication (a class can still be parameterized by other classes using modules).

The type of a class structure, $\texttt{sig}\ (\tau_y)\ \varphi\ \texttt{end}$, is composed of 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.

$$\gamma ::= \texttt{sig}\ (\tau)\ \varphi\ \texttt{end} \mid \tau \to \gamma$$
$$\varphi ::= \emptyset \mid \varphi\,;\texttt{field}\,u : \tau \mid \varphi\,;\texttt{method}\,m : \tau \mid \varphi\,;\texttt{super}\,s : \varphi$$
$$\sigma ::= \ \dots \mid \forall\,\bar\alpha.\,\gamma$$

In the concrete syntax, τ_y and φ are combined: methods that appear in τ_y but not is φ are flagged *virtual* (as they are not defined); other methods appear both in τ_y and φ, with the same type. When necessary, a type variable can also be bound to τ_y. For instance, the concrete syntax

```
sig (α) virtual copy : α method x : int end
```

expands to

$$\texttt{sig}\ (\texttt{rec}\ \alpha.\langle \texttt{copy} : \alpha; \texttt{getx} : \texttt{int}; \rho\rangle)\ \texttt{method getx} : \texttt{int}\ \texttt{end}.$$

Typing contexts are extended with class variable bindings and superclass bindings:

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

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 `field`, `method`, `super`, and `self` bindings have been removed. Typing rules are given in figure 3. We redefine $A^\star$ to be $A \setminus \{\texttt{field}, \texttt{self}, \texttt{super}\}$, so that superclass bindings are also removed. Generalization of class types $\texttt{Gen}(\gamma, A)$ is, as for regular types, $\forall\,\bar\alpha.\,\gamma$ where $\bar\alpha$ are all variables of γ that are not free in A.

Class bodies are typed by adding each component (inheritance clause, field, or method) one after the other. Fields are typed in $A^\star$, since other fields, self, and super bindings should not be visible in field expressions. On the contrary, methods may depend on all fields and super-classes that were previously defined (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 superclass variable is given the type of the superclass. Superclass 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-BODY and rule NEW.

When a value or method component is redefined, its type cannot be changed, since previously defined methods might have assumed the old type[2]. 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. Methods returning objects of the same type as self are thus correctly typed.

```
class duplicable () = struct
```

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

FIG. 2. Core class syntax

$$
\text{(Field)}\ \frac{A^\star \vdash a : \tau}{A \vdash \texttt{field}\ u = a : (\texttt{field}\ u : \tau)}
\qquad
\text{(Method)}\ \frac{A \vdash \texttt{self} : \langle m : \tau ; \tau' \rangle \qquad A \vdash a : \tau}{A \vdash \texttt{method}\ m = a : (\texttt{method}\ m : \tau)}
$$

$$
\text{(Inherit)}\ \frac{A^\star \vdash c : \texttt{sig}\ (\tau_y)\ \varphi\ \texttt{end} \qquad A \vdash \texttt{self} : \tau_y}{A \vdash \texttt{inherit}\ c\ \texttt{as}\ s : \varphi + (\texttt{super}\ s : \varphi)}
\qquad
\text{(Basic)}\ \frac{}{A \vdash \emptyset : \emptyset}
\qquad
\text{(Then)}\ \frac{A \vdash d : \varphi_1 \qquad A + (\varphi_1 \setminus \texttt{method}) \vdash b : \varphi_2}{A \vdash d\ ; b : (\varphi_1 \setminus \texttt{super}) \oplus \varphi_2}
$$

$$
\text{(Class-Body)}\ \frac{A^\star + \texttt{self} : \tau_y \vdash b : \varphi}{A \vdash \texttt{struct}\ b\ \texttt{end} : \texttt{sig}\ (\tau_y)\ \varphi\ \texttt{end}}
\qquad
\text{(New)}\ \frac{A \vdash c : \texttt{sig}\ (\tau_y)\ \varphi\ \texttt{end} \qquad \tau_y = \langle \texttt{method}\ (\varphi) \rangle}{A \vdash \texttt{new}\ c : \tau_y}
$$

$$
\text{(Super)}\ \frac{\texttt{super}\ s : \varphi \in A \qquad \texttt{method}\ m : \tau \in \varphi}{A \vdash s\#m : \tau}
\qquad
\text{(Object)}\ \frac{A^\star + \texttt{self} : \tau_y \vdash b : \varphi \qquad \tau_y = \langle \texttt{method}\ (\varphi) \rangle}{A \vdash \langle b \rangle : \tau_y}
\qquad
\text{(Class-Inst)}\ \frac{z : \forall \bar\alpha. \gamma \in A}{A \vdash z : \gamma[\bar\tau / \bar\alpha]}
$$

$$
\text{(Class-Fun)}\ \frac{A + x : \tau \vdash c : \gamma}{A \vdash \texttt{fun}\ (x)\ c : \tau \to \gamma}
\qquad
\text{(Class-App)}\ \frac{A \vdash c : \tau \to \gamma \qquad A \vdash a' : \tau}{A \vdash c\ a' : \gamma}
\qquad
\text{(Class-Let)}\ \frac{A \vdash c : \gamma \qquad A + z : \texttt{Gen}(\gamma, A) \vdash a : \tau}{A \vdash \texttt{class}\ z = c\ \texttt{in}\ a : \tau}
$$

FIG. 3. Typing rules for classes

```
  method copy = {⟨ ⟩}
end;;
class duplicable : unit → sig (α)
  method copy : α
end
```

In this class type, α is bound to the type of self. Thus, 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).

```
class duplicable_point x = struct
  inherit duplicable () inherit point x
end;;
class duplicable_point : int → sig (α)
  field x : int ref
  method copy : α
  method 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, CLASS-INST, CLASS-FUN and CLASS-APP are similar to the rules LET, INST, FUN and APP for core ML (described in appendix 1). The two rules CLASS-LET and CLASS-INST are essential since polymorphism of class types enables method specialization during inheritance, as explained above.

As an illustration of the typechecking rules we give a detailed derivation of the typing of the class scaled_point in the appendix 2.

4. Coercion

Polymorphism on row variables enables one 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 is not 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 one 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:

$$a ::= \ \ldots \mid (a : \tau <: \tau)$$

The corresponding typing rule is:

$$\text{(COERCE)}\quad \frac{\tau \leq \tau' \qquad A \vdash a : \theta(\tau)}{A \vdash (a : \tau <: \tau') : \theta(\tau')} \qquad \theta \text{ substitution}$$

The premise $\tau \leq \tau'$ means that τ is a subtype of τ'. As far as typechecking is concerned, we could have equivalently introduced coercions as a family of constants $(\_ : \tau <: \tau')$ of respective principal types $\forall \bar{\alpha}. \tau \to \tau'$ where $\bar{\alpha}$ are free variables of τ and τ' indexed by all pairs of types (τ, τ') such that $\tau \leq \tau'$.

The subtyping relation $\leq$ is standard [4]. We choose the simpler (and algorithmically more efficient) presentation of [16]. The constraint $\tau \leq \tau'$ is defined on regular trees as the smallest transitive relation that obeys the following rules:

Closure rules

$$\begin{array}{c} \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' \Longrightarrow \tau_1' \leq \tau_1 \wedge \tau_2 \leq \tau_2' \\ \langle \tau \rangle \leq \langle \tau' \rangle \Longrightarrow \tau \leq \tau' \\ (m : \tau_1; \tau_2) \leq (m : \tau_1'; \tau_2') \Longrightarrow \tau_1 \leq \tau_1' \wedge \tau_2 \leq \tau_2' \end{array}$$

Consistency rules

$$\begin{array}{c} \tau \leq \tau_1 \to \tau_2 \Longrightarrow \tau \text{ is of the shape } \tau_1' \to \tau_2' \\ \tau \leq \langle \tau_0 \rangle \Longrightarrow \tau \text{ is of the shape } \langle \tau_0' \rangle \\ \tau \leq (m : \tau_1; \tau_2) \Longrightarrow \tau \text{ is of the shape } (m : \tau_1'; \tau_2') \\ \tau \leq \emptyset \Longrightarrow \tau = \emptyset \\ \tau \leq \alpha \Longrightarrow \tau = \alpha, \end{array}$$

Our subtyping relation does not enhance subtyping assumptions on variables, and it is thus weaker than the subtyping relation used in [12], except on ground types.

For instance, the expression $\mathtt{fun}\ (x)\ x$ has type $\forall \alpha, \alpha' \mid \alpha \leq \alpha'. \alpha \to \alpha'$ in [12], while we can only type the equivalent expression $\mathtt{fun}\ (x)\ (x : \tau <: \tau')$ for particular instances (τ, τ') of (α, α') such that $\tau \leq \tau'$.

5. Semantics

We give a small step reduction semantics to our language. Values are of two kinds: regular expression values are either functions or object values. Class values are either class functions or reduced class structures. Object values and reduced class structures are composed of methods and fields which are themselves values; fields must precede methods and neither can be overridden in values. Values, evaluation contexts, and reduction rules are given in figure 4.

The first reduction rule shows that objects are just a restricted view of classes where instance variables have been hidden.

We have chosen to reduce inheritance in objects rather than classes. It would also be possible to reduce inheritance inside classes, and reorder methods and 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 performed in two steps, described by the four rules for objects: inheritance and evaluation of value components are reduced top-down (first rule, we remind that the meta-notation @ stands for the concatenation of sequences); the components are then re-ordered (last rule) and redundant components removed bottom-up (two middle rules).

The invocation of a method $\langle w \rangle \# m$ evaluates the corresponding expression $w(m)$ after replacing self, instance variables, and overriding by their current values. That is, the following substitutions are successively applied:

1. $[\langle w \rangle / \mathtt{self}]$ replaces $\mathtt{self}$ by $\langle w \rangle$,
2. $[w(u)/u]^{u \in dom\,(w)}$ replaces each outer instance variable u by its actual value. Inner instances of u, *i.e.* those appearing inside an object $\langle w' \rangle$, are not replaced since they are related to the inner object. Note that $w(u)$ is a value and does not contain free fields.
3. $[\langle w \mathbin{@} (\mathtt{field}\ u = {a_u}^{u \in V}) \rangle / \{\langle u = {a_u}^{u \in V} \rangle\}]^{V \subset U}$ replaces each outer occurrence of an overriding $\{\langle u = {a_u}^{u \in V} \rangle\}$ by a new object built from w by overriding fields $u \in V$ by $(\mathtt{field}\ u = a_u)^{u \in V}$. Inner occurrences, *i.e.* those appearing inside an object $\langle w' \rangle$, are not replaced since they are related to the inner object. Note that a_u is not necessarily a value, and may contain other outer overriding of fields, that should be replaced simultaneously, or equivalently in a bottom-up fashion (deeper occurrences being replaced first).

Coercion behaves as the 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:

$$\frac{A \vdash a : \tau \qquad \tau \leq \tau'}{A \vdash a : \tau'}\ \text{(SUB)}$$

This means that a well-typed expression that has been reduced may not always be typable without rule SUB. This is not surprising since explicit subtyping may disappear during reduction. Thus, implicit subtyping may be required after reduction. It is possible however to keep explicit subtyping information during reduction, and avoid the need for rule SUB. This would be obtained by replacing the rule

$$(a : \tau <: \tau') \longrightarrow a$$

by the following rules

$$\begin{array}{l} (v : \langle m_i : {\tau_i}^{i \in I} \rangle <: \langle m_i : {\tau_i'}^{i \in J} \rangle) \\ \qquad\qquad \longrightarrow \langle m_i = (v \# m_i : \tau_i <: \tau_i')^{i \in J} \rangle \\ (\mathtt{fun}\ (x)\ a : \tau_1 \to \tau_2 <: \tau_1' \to \tau_2') \\ \qquad\qquad \longrightarrow \mathtt{fun}\ (x)\ (a[(x : \tau_1' <: \tau_1)/x] : \tau_2 <: \tau_2') \end{array}$$

The counterpart is that types, although not actively participating, would be kept during reduction. The formulation we have chosen has a simpler semantics and makes it clearer that the reduction is actually untyped.

Values

$$
\begin{aligned}
v &::= \ldots \mid \texttt{fun}\ (x)\ a \mid \langle w \rangle \\
v_c &::= \texttt{fun}\ (x)\ c \mid \texttt{struct}\ w\ \texttt{end} \\
w &::= \emptyset \mid w_d\ ;\ w && \texttt{field}\text{ components preceed }\texttt{method}\text{ components, no overridings} \\
w_d &::= \texttt{method}\ m = a \mid \texttt{field}\ u = v
\end{aligned}
$$

Evaluation contexts

$$
\begin{aligned}
E &::= [] \mid \texttt{let}\ x = E\ \texttt{in}\ a \mid E\ a \mid v\ E \mid E\#m \mid \langle F \rangle \mid \texttt{new}\ E \mid \texttt{class}\ z = E_c\ \texttt{in}\ a \\
E_c &::= [] \mid E_c\ a \mid v_c\ E \mid \texttt{struct}\ F\ \texttt{end} \\
F &::= [] \mid F_d\ ;\ b \mid w_d\ ;\ F \\
F_d &::= \texttt{inherit}\ E_c\ \texttt{as}\ s \mid \texttt{field}\ u = E
\end{aligned}
$$

From classes to objects

$$\texttt{new}\ (\texttt{struct}\ w\ \texttt{end}) \longrightarrow \langle w \rangle$$

Reduction of objects

$$
\begin{aligned}
\texttt{inherit}\ (\texttt{struct}\ w\ \texttt{end})\ \texttt{as}\ s\ ;\ b &\longrightarrow w \mathbin{@} (b\,[w(m)/s\#m]^{m \in dom\,(w)}) \\
\texttt{field}\ u = v\ ;\ w &\longrightarrow w && \text{if } u \in dom\,(w) \\
\texttt{method}\ m = a\ ;\ w &\longrightarrow w && \text{if } m \in dom\,(w) \\
\texttt{method}\ m = a\ ;\ (\texttt{field}\ u = v\ ;\ w) &\longrightarrow \texttt{field}\ u = v\ ;\ (\texttt{method}\ m = a\ ;\ w)
\end{aligned}
$$

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

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

Reduction of coercions

$$(a : \tau <: \tau') \longrightarrow a$$

Reduction of other expressions

$$
\begin{aligned}
\texttt{let}\ x = v\ \texttt{in}\ a &\longrightarrow a[v/x] & \qquad \texttt{class}\ z = v\ \texttt{in}\ a &\longrightarrow a[v/z] \\
(\texttt{fun}\ (x)\ a)\ v &\longrightarrow a[v/x] & (\texttt{fun}\ (x)\ c)\ v &\longrightarrow c[v/x]
\end{aligned}
$$

Context reduction

$$
\begin{aligned}
E[a] &\longrightarrow E[a'] \text{ if } a \longrightarrow a' & \qquad E[b] &\longrightarrow E[b'] \text{ if } b \longrightarrow b' \\
E[c] &\longrightarrow E[c'] \text{ if } c \longrightarrow c'
\end{aligned}
$$

FIG. 4. Semantics of Objective ML

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

Theorem 1 (Subject Reduction) *Reduction preserves typings* (i.e. for any A, if $A^\star \vdash a : \tau$ and $a \longrightarrow a'$ then $A^\star \vdash a' : \tau$.)

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.)

See appendix 4 for proofs of these theorems.

These results easily extend to cope with constants, as in core ML, provided δ-rules for constants are consistent with their principal types.

6. Type inference

Types of Objective ML are a restriction of record types. First-order unification for record types is decidable, and solvable unification problems admit principal solutions, even in the presence of recursion [31].

The unification algorithm is a simplification of the one used in ML-ART [31]. It is described in figure 5 as a rewriting process over unification problems. This formalism was introduced in [15] and has already been used for record types in [30]. A unification problem also called a *unificand*, is a multi-set of multi-equations preceded by a list of existentially quantified variables. It is written $\exists\,\alpha_1, \ldots \alpha_p.\, e_1 \wedge \ldots e_q$. A *multi-equation* e is a multi-set of types written $\tau_1 \doteq \ldots \tau_n$. The algorithm assumes that recursive types $\mu\alpha.\tau$ have been encoded using equations $\exists\,\alpha.\, \alpha \doteq \tau$.

A substitution is a solution of a multi-equation if it makes all its types equal. A solution of a unificand is the restriction of a common solution to all its multi-equations outside of the existentially quantified variables.

$$\frac{\alpha \doteq e \wedge \alpha \doteq e'}{\alpha \doteq e \doteq e'} \text{ (FUSE)} \qquad \frac{f(\alpha_i{}^{i \in I}) \doteq f(\alpha_i'{}^{i \in I}) \doteq e}{f(\alpha_i{}^{i \in I}) \doteq e \wedge (\alpha_i \doteq \alpha_i')^{i \in I}} \text{ (DECOMPOSE) (1)} \qquad \frac{e[\tau/\alpha] \qquad \alpha \notin \tau}{\exists\, \alpha .\, e \wedge \alpha \doteq \tau} \text{ (GENERALIZE) (2)}$$

$$\frac{(m_1 : \alpha_1 ; \alpha_1') \doteq (m_2 : \alpha_2 ; \alpha_2') \doteq e}{\exists\, \alpha' .\, (m_2 : \alpha_2 ; \alpha_2') \doteq e \wedge \alpha_1' \doteq (m_2 : \alpha_2 ; \alpha') \wedge \alpha_2' \doteq (m_1 : \alpha_1 ; \alpha')} \text{ (MUTATE)}$$

(1) In Rule DECOMPOSE, f is any type symbol, including $(m : \_ ; \_)$ as well.

(2) To ensure termination, rule GENERALIZE must be restricted to the case where τ is not a variable and α appears in e but not as a term variable of e.

FIG. 5. Unification as solving multi-sets of multi-equations

Unificands can be simplified by applying the rewriting rules given in figure 5. Structural rules have been omitted: they include associativity and commutativity of both $\wedge$ and $\doteq$ and the extrusion and renaming of existential variables. Rules FUSE, DECOMPOSE and GENERALIZE are standard. Rule FUSE merges two multi-equations that have a variable in common. Rule DECOMPOSE decomposes terms of a multi-equations into smaller ones. Rule GENERALIZE splits terms into smaller terms. Thus, unificands can always be rewritten so that terms are of depth at most one. This permits maximal sharing during unification. It also ensures termination of rewriting in the presence of recursive types. The only difference with unification in a free algebra is the mutation rule MUTE for left-commutativity. It identifies two terms $(m_1 : \tau_1 ; \tau_1')$ and $(m_2 : \tau_2 ; \tau_2')$ with different top symbols $(m_1 : \_ ; \_)$ and $(m_2 : \_ ; \_)$ provided their equality can be established by the application of an axiom at the root.

The algorithm proceeds by rewriting multi-sets of multi-equations according to the above rules. Each step preserves the set of solutions. Moreover, the process always terminates, reducing any unificand to a canonical form.

A unificand is in a solved form if all of its multi-equations are merged and each of them is fully decomposed (*i.e.* it contains at most one non-variable term). Principal unifiers can be read directly from solved forms. A canonical unificand that is not in a solved formed contains a clash (two incompatible types that should be identified) and is not solvable.

The framework and the meta-theory of unificands are standard. The equational theory of object types is a sub-case of the more general algebra of records types; for details and proofs, the reader is referred to [30].

Objective ML does not allow classes as first-class values. Indeed, in the expression `fun` (x) a, variable x cannot be bound to a class (or a value containing a class). Thus, class types never need to be guessed. Polymorphism is only introduced at LET bindings of classes or values. This ensures that type inference reduces to first-order unification, as it is the case in ML. Consequently, Objective ML has the principal type property. Type inference for classes is straightforward. The links between first-order unification, type inference and principal types are described in a more general setting in [29].

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. Abbreviation enhancements

Object types tend to be very large. Indeed, the type of an object lists all its methods with their types, which can themselves contain other object types. This quickly becomes unmanageable [31, 11]. Introducing abbreviations is thus of crucial importance. This section presents the general abbreviation mechanism of Objective ML and the next section focuses on abbreviating object types. The simple type abbreviation mechanism of ML is not sufficiently powerful: abbreviations are expanded and lost during unification and they do not interact well with recursive types. Several improvements have thus been made to the abbreviation mechanism. First, abbreviations are kept during unification and propagated as much as possible. Second, a larger class of abbreviations are accepted: abbreviations can be recursive and their arguments can be constrained to be instances of some given types.

In our implementation, types are considered as graphs. In particular, when two types are unified, they become identical rather than two separate, equal types. A construct has been added to the syntax to express type graphs: the construct $(\tau \texttt{ as } \alpha)$ is used to bind α to τ, similarly to the notation $\texttt{rec } \alpha.\tau$. However, a main difference is that with aliases α is also bound outside of τ. As an example, 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. There are two reasons for considering types as graphs. First, unification rolls types. For instance, unifying types $\tau = \alpha$ and $\tau' = \langle m : \alpha \rangle$ results in type $\tau = \tau' = (\langle m : \alpha \rangle \texttt{ as } \alpha)$, rather than instantiating α to $\langle m : \alpha' \rangle \texttt{ as } \alpha'$ in both types (in the later case, τ' would become $\langle m : \langle m : \alpha' \rangle \texttt{ as } \alpha' \rangle$). Second, unification propagates abbreviations. Abbreviations can be considered as names for nodes. Unifying an abbreviated type with another type makes both types abbreviated. For instance, unifying the argument of a functional type to an abbreviated type may propagate the abbreviation to the result type. This is demonstrated in the following example.

```
let bump x = x#move 1; x;;
value bump :
  (⟨ move : int → β; .. ⟩ as α) → α =
  ⟨fun⟩
```

Nodes are shared between the argument type and the result type. The ellipsis stands for an anonymous row variable. When typing the expression `bump p` below, type (⟨`move : int` $\rightarrow$ β; ..⟩ `as` α) and type `point` are identified. The type of `bump p` is thus also abbreviated to `point`.

```
let p = new point 7;;
value p : point = ⟨obj⟩
bump p;;
- : point = ⟨obj⟩
```

Not all the sharing is exposed to the user : sharing reveals too much useless information. So, only aliasing of open object types (thus row variables can be printed as ellipses) and aliasing defining recursive types are printed. It would be possible to remove some aliasing during type generalization, so that printed types would exactly reflect their internal representations. However, this would complicate the implementation needlessly.

Abbreviations can be recursive. That is, in the definition of the abbreviation `type` $(\bar{\alpha})\ \kappa = \tau$, the type constructor κ may occur in the body τ, as long as all occurrences have the same parameters $\bar{\alpha}$. This restriction is extended to mutually recursive abbreviations. It ensures that abbreviations expand to regular trees. In the implementation, any type constructor standing for an abbreviation caches the expansions of abbreviations it appears in. Thus, when an abbreviation is expanded several times during the traversal of a type, it expands each time to the same type.

Type abbreviations are generalized to allow constraints on the type parameters of the abbreviations. This is an extension to the abbreviations of LCS [5], that were also used in [31]. In an abbreviation definition, parameters are types rather than type variables: `type` $(\bar{\tau})\ \kappa = \tau_0$. All free variables of τ must be bound in $\bar{\tau}$. Actual arguments of an abbreviation must always be instances $\theta(\bar{\tau})$ (for some substitution θ) of the parameters $\bar{\tau}$. Then, the abbreviation can expand to type $\theta(\tau_0)$. For instance, if the type constructor κ is defined as `type` $(\alpha * \alpha')\ \kappa = \alpha \rightarrow \alpha'$, then $(\texttt{int} * \texttt{bool})\ \kappa$ will expand to $\texttt{int} \rightarrow \texttt{bool}$. To expand an abbreviation, the arguments are usually substituted for the parameters. Instead, we choose to unify the arguments with the corresponding parameters. The constraints need only to be enforced when parsing a type given by the user. Then, expansion is guaranteed to succeed. Indeed, a substitution θ can always be applied to an abbreviation $(\bar{\tau})\ \kappa$. The expansion of $\theta((\bar{\tau})\ \kappa)$ is equal to the result of applying the substitution θ to the expansion of $(\bar{\tau})\ \kappa$. In particular, constraints are preserved by substitution.

8. Abbreviating object types

We will now describe how the abbreviation mechanism presented in the previous section is used to generate abbreviations for objects. This mechanism is used to automatically abbreviate object constructors: the expression `new` z will have type $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow (\tau_i')\ \kappa_z$, where κ_z is the abbreviation associated with class z.

General type abbreviations, introduced in the previous section, can be used to simplify object types. Rather than sorting types to ensure that object types are well-formed, we require the stronger condition that any two object types that share the same row variable must be equal. This eliminates 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, no primitive operation on objects exhibits such a type. These types can thus be ruled out without seriously restricting the language. Moreover, all programs keep the same principal types. This restriction was implemented to avoid explaining sorts to the user. It also makes the syntax for types somewhat clearer, as row variables can then always be replaced by ellipsis. Furthermore, sharing can still be described with aliasing. For instance, $\langle m : \tau; \rho\rangle \rightarrow \langle m : \tau; \rho\rangle$ is written $(\langle m : \tau; ..\rangle\ \texttt{as}\ \alpha) \rightarrow \alpha$.

A class definition `class` $z = c$ `in` ... automatically generates an abbreviation for the type of its instances. For specifying it, one actually needs to add type parameters to the class definitions, corresponding to the one of the abbreviation. That is, we should write

$$\texttt{class}\ (\bar{\alpha})\ z = c\ \texttt{in} \ldots \tag{1}$$

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

In fact, abbreviations are generated from class *types*. It follows from type inference that the class definition c has a principal class type $\tau_0' \rightarrow \ldots \rightarrow \tau_n' \rightarrow \texttt{sig}\ (\tau_y)\ \varphi\ \texttt{end}$. 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 `method` (φ) 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 `method` (φ) is exactly $(m_i : \tau_i)^{i \in I}$, 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 two abbreviations:

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

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 $\texttt{int} \rightarrow \texttt{sig}\ (\langle \texttt{move} : \texttt{int} \rightarrow \texttt{int}; \rho\rangle)\ \varphi\ \texttt{end}$ for some φ whose only method is $\texttt{move} : \texttt{int} \rightarrow \texttt{int}$. Thus, class point is not virtual. The two following abbreviations are generated 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 the type `point` is indeed an abbreviation for the type of objects of the class `point`, and that the type of an object of any subclass of the class `point` is an instance of the type $\rho\ \#\texttt{point}$.

In the concrete syntax, the row variable ρ is treated anonymously (as an ellipsis) and is omitted. The former abbreviation $\#\kappa_z$ is given a lower priority than the regular ones in case of a clash. It also 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. The previous definitions can be rewritten to handle the general case correctly.

Constrained abbreviations are natural for abbreviating objects, as, for instance, a sorted list of comparable objects should be parameterized by the type of its elements, which in turn 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).

Constrained type abbreviations are also convenient since, in a class definition $\texttt{class}\ (\bar{\alpha})\ z = c\ \texttt{in} \ldots$, class type parameters $\bar{\alpha}$ may have been instantiated to some types $\bar{\tau}_\alpha$ while inferring the class type $\tau'_0 \rightarrow \ldots \rightarrow \tau'_n \rightarrow \texttt{sig}\ (\tau_y)\ \varphi\ \texttt{end}$. The two abbreviations generated by the class definition are thus:

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

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 of their type parameters. The parameters that constraint the type abbreviations are described by constraint clauses:

```
class α circle (p : α) = struct
  field point = p
  method center = point
  method move m =
    if m = 0 then 0 else
    point#move (1 + Random.int m)
end;;
class α circle : α → sig
  constraint α = ⟨ move : int → int; .. ⟩
  field point : α
  method center : α
  method move : int → int
end
```

This class defines the abbreviation

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

As a result of the abbreviation mechanisms, type inference may reject some class definitions whose principal types have free variables. For instance, the following variant of class `point` is rejected, since the method `getx` is polymorphic and therefore the class should be parametric.

```
class point x0 = struct
  field x = x0
  method getx = x
end;;
```

Of course, one could choose an arbitrary ground class type, for instance:

```
class point : int → sig
  field x : int
  method getx : int
end
```

Any other ground type could be used instead of `int`. We decide to reject those programs. This preserves the property that any typable program has a principal type —and all other useful properties of the type system.

This phenomenon is not new. It already appeared in several extensions of ML. Imperative constructs limit polymorphism. Thus, some variables that are not generalizable may occur in the type of a top level expression. In such a case, most languages would reject the program. For instance, the extension to ML with dynamics [20] rejects `fun x → dynamics x`, since the dynamic type of `x` in `dynamics x` is statically unknown.

All the examples above would have principal types as long as type inference is concerned. We can argue that some programs have been rejected for sake of simplicity and uniformity of the language, but not because of a failure of type inference: For instance, in Objective ML we could just omit the corresponding abbreviation whenever some type parameter is missing, and print a warning message instead of an error message.

9. 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.

Before we explore these extensions, let us consider an interesting restriction of the language. If recursive types are only allowed when the recursion traverses an object type, Objective ML becomes a conservative extension of ML, which we claimed in the introduction. Of course, all ML programs can be defined, and behave similarly. Moreover, programs that are syntactically ML programs are now well-typed ML programs if and only if they are well-typed in Objective ML. However, in the implementation Objective Caml, the presence of modules requires the use of recursive abstract types as well. This is because recursive object types may be abstracted. Thus, Objective Caml is not strictly speaking a conservative extension of ML. Still, it is a conservative extension of ML with recursive types.

9.1. Imperative features

We have intentionally 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 put emphasis on objects and classes. The addition of imperative features to Objective ML is theoretically as simple and as useful practically as their addition to ML. Both the semantics and the properties of reduction with respect to typing extend to operations on the store without any problem. The formalization copies the one for core ML.

In fact, the implementation Objective Caml also allows fields to be mutable in a similar way mutable record fields are treated in Caml [21]. For instance, we could have written:

```
class point x0 = struct
  field mutable x =  x0
  method move d = (x ← x + d; x)
end;;
class point : int → sig
  field mutable x : int
  method 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). Mutable fields in classes are typed as any other fields, except that mutability properties are also checked during typechecking.

9.2. Local bindings

As shown by the evaluation rules for objects, both value and method components are bound to their rightmost definitions. All value components 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 a field can be restricted so that the field is no more 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 usually expects from private fields, as a class cannot, for instance, inherit a field and hide it from its subclasses (see section 10.1).

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 fields. 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

```
class point y = struct method x = y end;;
```

could be automatically transformed into the equivalent program:

```
class point y = struct
  local y = y in method 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 flattened, and method overriding is resolved.

9.3. Coercion primitives

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

For convenience, we introduce a collection of coercion primitives:

$$(\_ <: \tau) : \forall\,\bar\alpha.\, S(\tau) \rightarrow \tau$$

where $\bar\alpha$ 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 from the left hand side. (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 pairs 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,

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

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

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

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 latter 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 $\langle m : \mathtt{int}\rangle \rightarrow \mathtt{int}$. There are only two solutions, $\langle m : \mathtt{int}\rangle \rightarrow \mathtt{int}$ and $\langle\rangle \rightarrow \mathtt{int}$ 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. For instance, if τ is $\langle x : \alpha\rangle \texttt{ as } \alpha$, then both $\langle x : \tau; \rho\rangle$ and $\langle x : \beta; \rho'\rangle \texttt{ as } \beta$ are solutions for $S(\tau)$, but no solution is more general than both of these. Our choice of S (and correspondingly, our choice of coercion primitives) is somehow arbitrary, but works well in practice. This justifies the exclusion of semi-explicit coercions from the core language, but leave them as a collection of primitives. In fact, most coercions are of the form $(a : S(\tau) <: \tau)$. Thus, the domain of a coercion rarely needs to be given.

10. Future work

This short section describes three possible extensions of importance to Objective ML. Each extension requires further theoretical and design investigation before it can be integrated within Objective Caml.

10.1. Restriction of class interfaces

In section 9.2 we have shown that field components can be declared local to a class. However, this does not enable class components to be hidden *a posteriori*. Assume, for instance, that a library provides an implementation of a class z with two fields x and x' and two methods m and m'. A module may define a class z'' that inherits from an imported class z' whose interface is a restriction of the one of the class z to the field x and the method m only. Can class z be used as an import to the module? This problem corresponds to a common situation of interface restriction when reusing code. However, interface restriction is not currently possible.

Private fields would actually not be difficult to hide. However, hiding methods in subclasses conflicts with late binding and a flat method name space. For instance, assume, method m' is implicitly hidden when inherited in class z'', and that class z'' defines a method m', possibly with another type!

Clearly, when a method m is hidden in a class z, self-invocations of m in all other methods of z should be replaced by calls to a function representing the method m. This is a complex operation that is difficult to compile.

Another problem is that method m' appears in the type of `self`. Hiding the method thus requires to modify *a posteriori* the type of `self`. This would not be correct if, for instance, this type is the type of a method argument.

A partial solution is to give each method a different view of `self` inside classes. This is usually the case when classes are treated as a collection of pre-methods. Another choice, weaker but still useful, is to split the input and output view of `self`. The former lists the methods that are required while the latter enumerates methods that are provided. However, in the presence of type inference, such solutions tend to increase the size of a class to a point that may become unreadable [31]. The gain in expressiveness is also weakened by a later detection of errors. Clearly, it is an error if a method has incompatible required and provided types. However, this would only be detected when the object is created. In the design of Objective ML, we have deliberately limited the expressiveness of class types to keep them readable. Many variations are theoretically possible, but very few of them seem to improve expressiveness significantly without sacrificing simplicity.

Another possibility is to introduce private methods. They would not appear in the type of `self`, consequently, they should be invoked differently. Private methods could have the same scope as fields. In particular, they could be hidden *a posteriori* as well.

The addition of *final* classes could also resolve the problem. These classes could not be inherited. Then, a class could be soundly matched against a final class interface that omits some of its methods.

10.2. Polymorphic methods

In a classical programming style, functions and data are clearly separated. Functions are often polymorphic and thus can be applied uniformly to different kinds of data. Data may be structured. It very rarely carries functions, and is usually monomorphic. In objects, data and methods are jointly defined and stored or passed as arguments together — at least from a theoretical point of view.

Let-bound top level functions often become methods of λ-bound first-class objects. Unfortunately, polymorphism is lost during this transformation. For instance, a class implementing sets, would naturally provide a fold method. The inferred class type would be of the form:

```
class α set = struct  ...
  method fold : (α → β → β) → β → β
end
```

However, this is rejected, since variable β is unbound in α `set`. An attempt to fix the problem would be to parameterize the class `set` over β as well, that is, to replace α `set` in the definition above by (α, β) `set`. However, this is not very intuitive, since the object stays parametric in β even when all its fields have a ground type. Moreover, the method `fold` becomes monomorphic and thus can only be applied to functions of the same type, whenever the object is λ-bound.

The intuition is of course that the method `fold` should be polymorphic. That is, the class `set` should have the following class type:

```
class α set = struct ...
  method fold : All β. (α → β → β) → β → β
end
```

The addition of polymorphic methods could also be used to reduce the number of explicit coercions. In a class definition methods may have types more polymorphic than expected. For instance, assume that class point has type:

```
class point (int) = struct
   field x : int  method getx : int
end;;
```

Then, the following subclass of `point` will not typecheck:

```
class eq_point x = struct
  inherit point x
  method eq p = p#getx = self#getx
end;;
```

The parameter `p` of the method `eq` does not need to be a point but an object with method `getx` of type `int`. Thus, its type ⟨`getx : int; ..`⟩ → `bool` has a free row variable. As for the case of `set`, the row variable in the type of `p` can be bound in in a constraint type parameter as follows:

```
class α eq_point x = struct
 inherit point x
 method eq (p:α) = p#getx = self#getx
end;;
class α eq_point : int → sig
  constraint α = ⟨ getx : int; .. ⟩
  field x : int
  method getx : int
  method eq : α → bool
end
```

Again, this is not very intuitive and one might prefer to add a stronger type constraint. One choice is to require `p` to be of the same type as `self`. However, this unnecessarily makes `eq` a binary method and so restricts its further use with arguments of type `eq_point` only. Constraining `p` to be a `point` in the definition of the method `eq` is another possibility:

```
class eq_point x = struct
 inherit point x
 method eq (p:point) = p#getx = self#getx
end;;
```

```
class eq_point : int → sig
  field x : int
  method getx : int
  method eq : point → bool
end
```

This solution is more general, although it usually requires explicit coercion when invoking the method `eq`:

```
let p = eq_point 1 in p#eq (p ⟨: point);;
```

Polymorphic methods would allow a more natural class type for the `eq_point` (first definition):

```
class eq_point : int → sig
  field x : int
  method getx : int
  method eq p :
      All (⟨getx : int; ..⟩ as α). α → bool
end;;
```

Moreover, thanks to the polymorphic (anonymous) row variable, messages could then be sent to the method `eq` with an argument of type either `point` or `eq_point`.

We consider that the lack of polymorphic methods is a weakness of Objective ML. We believe that polymorphic methods would make most explicit coercions unnecessary.

Some solutions to extend ML with first class-polymorphism already exist in the literature. Simple but rudimentary proposals can be found in [31, 24] and better integration of first-class polymorphism inside Objective ML has recently been studied in [14].

10.3. Integrating classes and modules

Objects and classes of Objective ML are orthogonal to the other extensions of ML. In particular, the module system of ML extends directly to classes and objects [18]. Indeed, the implementation of Objective ML, called Objective Caml [19], offers a rich language of both modules and classes. Classes and modules share a lot of properties: they offer some form of abstraction; they also help structuring large applications; and they facilitate reusability of code. In fact, they are quite different. Modules are a very general and powerful abstraction. However, it is difficult to allow recursion between several modules or to give a meaning to self inside modules. On the other hand, classes are a much more specialized paradigm that has proved extremely convenient for some applications. Objects find their limitation with multiple dispatch. Hiding components also remains a difficult task.

For historical reasons, libraries of Objective Caml are implemented as modules. In practice, many of these libraries could be rewritten as classes. Choosing one style or another is not insignificant, since it is a global commitment to the architecture of the application. The class version and the module version of the same libraries are very similar, but their code cannot currently be shared. This is, of course, unsatisfactory. We hope that more work will allow a better integration of modules and classes.

11. Comparison to other works

The work closest to Objective ML is ML-ART [31]. Here, object types are also based on record types and have similar 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 is 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 values in ML-ART. We, however, do not think this is a major advantage. The restriction is a deliberate choice in the design of Objective ML, to keep the language simpler. In theory, most features of ML-ART could have been kept in Objective ML. In practice, however, it would have changed the language significantly.

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 [31]). We found that this extra flexibility is not worth the complication of class types.

Our object types are a simplification of those used in [32]. 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, our 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 kinded record types introduced in [25]. Ohori also proposed an efficient compilation of polymorphic records (which does not scale up to extensible records) in [26]. However, his approach, based on the correspondence between types and domains of records cannot be applied to the compilation of objects with code-free coercions.

Objects have been widely studied in languages with higher-order types [9, 23, 7, 2, 28, 6]. These 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.

Our calculus differs significantly from Abadi's and Cardelli's primitive calculus of objects mostly as a result of design choices. We have chosen primitive classes because inferred types of sets of pre-methods would be too complex to be readable (see [31] for instance). We have emphasized the role of row variables because we have chosen not to infer subtyping, therefore avoiding the more complicated framework of constraint types. On the other hand we have included other features such as instance variables, to avoid their encoding as methods not involving self, and to keep with the more simple state-abstraction mechanism by scope hiding. Technically a major difference, Objective ML does not allow method overriding.

Open record types are connected to the notion of matching introduced by Kim Bruce [7, 8]. Matching seems to be at least as important as subtyping in object-oriented languages. Row variables in object types express 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 [27] for a first-order version of Abadi and Cardelli's calculus of primitive objects [1]. However, that language is missing important features from the higher-order version [2]. Type inference is based on subtyping constraints and the technique is similar to the one used in [11]. This latter proposal [11, 12] 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 too large. 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.

The remainder of this section is dedicated to the comparison with three other proposals for adding objects to ML. They all use implicit subtyping, which is, however, restricted to atomic structural subtyping [22, 13]. As a result, they all have the same difficulty with parameterized classes, making it impossible to relate objects created from classes with a different number of parameters, even when the objects have the same interface. For instance, objects of a class `string` are of incompatible type with objects of a parameterized class `vector` when the parameter type is character. In Objective ML, such objects could be mixed.

In [6], Bourdoncle and Metz propose a language based on some restricted form of type constraints [12]. However, they do not provide type inference.

The two following proposals include type inference; however, fully polymorphic method invocation cannot be typed. Two different solutions are proposed; they both amount to providing some explicit type information at method invocation.

More precisely, in Duggan's proposal [10], methods must be predeclared with a particular type scheme. Thus methods carry type information like 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 a variable 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`$\rangle$. The user must provide more type information that in Objective ML. The same method name cannot be used in two different objects with unrelated types. Objects of parameterized classes are treated especially, using constructor kinds. As mentioned above, objects of a parameterized class reveal forever that they are parameterized. For instance, let us 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 of the type τ_y is hidden. This reveals the dependence of τ_y on its parameters, and the parameters themselves. As explained above, methods of parameterized classes are incompatible with methods of non-parameterized classes. Conversely, Objective ML does not currently allow polymorphic methods while Duggan's proposal does. 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 in [17, 31, 24]. Recursive kinds actually allow some form of polymorphism that is different from polymorphic methods discussed in section 10.

In Object ML [34], Reppy and Riecke treat objects as a generalized form of concrete data-types. Types are also inferred in Object ML, but the authors do not claim a principal type property. Also, method invocation must always mention the class of the object to which the method belongs. Each object is actually tagged with a constructor that carries the class the object originated from. Therefore, objects can be tested for membership to some arbitrary class in some inheritance relationship. Only single inheritance is allowed. The subtyping relationship between objects is declared and corresponds to the inheritance forest. Classes are generative, that is, objects of different classes have different types. Although these types can be related by subtyping, they are never in an instance relationship. Some object coercions, but apparently not all, are implicit. On the contrary in Objective ML, classes are transparent, that is, objects types are structural and only describe the interface of objects: two objects with exactly the same interface have equal types. Two objects of classes in a subclass relationship are not necessarily related, but when they are, one type is simply an instance of the other. Object ML does not provide any inheritance mechanism, except by means of encodings [33]. Typing of binary methods is also a problem, which is solved via runtime class-type tests.

Conclusion

Objective ML has been designed to be the core of a real programming language. Indeed, the constructs presented here have been implemented in Objective Caml. We chose class-based objects since this approach is now well understood in a type framework and it does not require 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 [31, 11]. 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 [35, 32]. 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. Specifically, 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. That should be particularly interesting to compare these two styles of large-scale programming, and help us to better integrate them. This is an important direction for future work.

Acknowledgments

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

Notes

1. The syntax has been slightly modified here in order to keep the concrete syntax and the abstract syntax closer.
2. One may imagine relaxing this constraint, and allow the type of the redefined method to be a subtype of the original method. One would, however, lose a property we believe important: rule INHERIT shows that the type a class gives to self is a common instance of the different types of self in its ancestors; as a consequence, the type of self in a class unifies with the type of any object of a subclass of this class.

References

[1] 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.

[2] 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.

[3] Martín Abadi and Luca Cardelli. *A theory of objects*. Springer, 1996.

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

[5] 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.

[6] François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *Proceedings of the 24th ACM Conference on Principles of Programming Languages*, pages 302–315, July 1997.

[7] Kim B. Bruce. Typing in object-oriented languages: Achieving expressiveness and safety. To appear.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In *ESOP '88*, volume 300 of *Lecture Notes in Computer Science*, pages 94–114. Springer Verlag, 1988.

[14] Jacques Garrigue and Didier Rémy. Extending ML with semi-explicit higher-order polymorphism. In *International Symposium on Theoretical Aspects of Computer Software*, Japan, September 1997.

[15] Claude Kirchner and Jean-Pierre Jouannaud. Solving equations in abstract algebras: a rule-based survey of unification. Research Report 561, Université de Paris Sud, Orsay, France, April 1990.

[16] 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.

[17] 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.

[18] Xavier Leroy. A modular module system. Research report 2866, INRIA, April 1996.

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

[20] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

[21] Xavier Leroy and Pierre Weis. *Manuel de référence du langage Caml*. InterÉditions, 1993.

[22] John C. Mitchell. Coercion and type inference. In *Eleventh Annual Symposium on Principles Of Programming Languages*, 1984.

[23] 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.

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

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

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

[27] 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.

[28] 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".

[29] Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatisme, BP 105, F-78 153 Le Chesnay Cedex, 1992.

[30] 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.

[31] 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.

[32] 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.

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

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

[35] Mitchell Wand. Complete type inference for simple objects. In D. Gries, editor, *Second Symposium on Logic In Computer Science*, pages 207–276, Ithaca, New York, June 1987. IEEE Computer Society Press.

Appendices

1. 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.

2. An example of typing derivation

In this section, we give the typing derivation for class `scaled_point`. Our focus here is not to explain type inference, but simply to illustrate the typing rules.

We assume that the class `point` has already been typed, that is, we type `scaled_point` in the environment A_0 containing the following class-type (we use `#point` as an abbreviation for $\langle move : \texttt{int} \to \texttt{int}; ..\rangle$):

```
int →
    sig (#point)
        field x : int ref
        method move : int → int
     end
```

We remind the definition of class `scaled_point`:

```
fun (s0)
    struct
        inherit point 0 as parent;
        field s = s0;
        method scale = s;
        method move =
            fun (d) parent#move(d * self#scale)
     end
```

The remainder of this section is a proof that class `scaled_point` has the following class type (we use `#scaled_point` is an abbreviation for $\langle move : \texttt{int} \to \texttt{int}; scale : \texttt{int}; ..\rangle$):

```
int →
    sig (#scaled_point)
        field x : int;
        field s : int;
        method move : int → int;
        method scale : int
     end
```

Let A_1 for A_0 extended with $s_0 : \texttt{int}$ and A_2 be A_1 extended with $\texttt{self} : \texttt{\#scaled\_point}$. The body of the inheritance clause must be typed in $A_2^\star$ which is equal to A_1. By rule CLASS-INST we have:

```
A1 ⊢ point :
    int →
        sig (#scaled_point)
            field x : int ref
            method move : int → int
         end
```

Note that we have chosen an instance of the type of class `point` where self type is `#scaled_point` (an instance of type `#point`). Thus, by rule CLASS-APP, we have:

```
A1 ⊢ point 0 :
    sig (#scaled_point)
        field x : int ref
        method move : int → int
     end
```

Applying rule INHERITS we get:

```
A2 ⊢ inherit point 0 as parent :
        (field x : int;
         method move : int → int;
         super parent :
             (field x : int;
              method move : int → int))
```
(1)

The rest of the class body must be typed in environment A_3 equal to A_2 extended with

```
field x : int; super parent :
    (field x : int; method move : int → int)
```

Since $A_3^\star$ is A_1, we have $A_3^\star \vdash s_0 : \texttt{int}$, and by rule FIELD,

$$A_3 \vdash \texttt{field}\ s = s_0 : \texttt{field}\ s : \texttt{int}. \tag{2}$$

The rest of the class body must be typed in A_4 equal to A_3 extended with $\texttt{field}\ s : \texttt{int}$. Since $A_4 \vdash s : \texttt{int}$, we have by rule METHOD

$$A_4 \vdash \texttt{method}\ scale = s : \texttt{method}\ scale : \texttt{int}. \tag{3}$$

Using rules SEND and SUPER, we also have $A_4 \vdash a : \texttt{int} \to \texttt{int}$ where

$$a \stackrel{def}{=\!=} \texttt{fun}\ (d)\ parent\#move(d * \texttt{self}\#scale)$$

Thus,

$$A_4 \vdash \texttt{method}\ move = a : \texttt{method}\ move : \texttt{int} \to \texttt{int}.$$

By rule THEN applied to (3) and the previous judgment, we have

$$A_4 \vdash (\texttt{method}\ scale = s; \texttt{method}\ move = a) : (\texttt{method}\ scale : \texttt{int}; \texttt{method}\ move : \texttt{int} \to \texttt{int})$$

By rule THEN gain, applied to (2) and the previous judgement, we have

$$A_3 \vdash (\texttt{field}\ s = s_0; \texttt{method}\ scale = s; \texttt{method}\ move = a) : (\texttt{field}\ s : \texttt{int}; \texttt{method}\ scale : \texttt{int}; \texttt{method}\ move : \texttt{int} \to \texttt{int})$$

Hence, by rule THEN again applied to (1) and the previous judgement, we have $A_2 \vdash b : \varphi$ where

$$b \stackrel{def}{=\!=} (\texttt{inherit point}\ 0\ \texttt{as}\ parent; \texttt{field}\ s = s_0; \texttt{method}\ scale = s; \texttt{method}\ move = a)$$

$$\varphi \stackrel{def}{=\!=} (\texttt{field}\ x : \texttt{int}; \texttt{method}\ move : \texttt{int} \to \texttt{int}; \texttt{field}\ s : \texttt{int}; \texttt{method}\ scale : \texttt{int})$$

Since $A_2 \vdash \texttt{self} : \texttt{\#scaled\_point}$, applying rule CLASS-BODY leads to:

$$A_1 \vdash \texttt{struct}\ b\ \texttt{end} : \texttt{sig}\ (\texttt{\#scaled\_point})\ \varphi\ \texttt{end}$$

Finally, by rule CLASS-FUN, we get:

$$A_0 \vdash \texttt{fun}\ (s_0)\ \texttt{struct}\ b\ \texttt{end} : \texttt{int} \to \texttt{sig}\ (\texttt{\#scaled\_point})\ \varphi\ \texttt{end}$$

3. Binary methods

In Objective ML, it is possible to define binary methods, that is, methods that receive as a parameter an object of the same type as self. Furthermore, a class that has binary methods can be freely extended by inheritance. Of course, binary methods remains binary in a subclass.

The virtual class `comparable` is a template for classes with a binary method `leq`. The component `virtual leq` is a type constraint on the type of self. This method must be applied to an object of the same type as self.

```
class comparable () = struct virtual (α)
  virtual leq : α → bool
end;;
class comparable : unit → sig virtual (α)
  virtual leq : α → bool
end
```

Class `int_comparable` inherits from class `comparable`. It implements method `leq` and adds a method `getx`.

```
class int_comparable (x : int) = struct
  inherit comparable ()
  field x = ref x
  method getx = !x
  method leq o = !x ≤ o#getx
end;;
class int_comparable : int → sig (α)
  field x : int ref
  method leq : α → bool
  method getx : int
end
```

Method `leq` still expects to be applied to an object of the same type as self. So, type `int_comparable` = `rec` $\alpha.\langle$`leq` : $\alpha \to$ `bool`; `getx` : `int`$\rangle$ is not a subtype of type `comparable` = `rec` $\alpha.\langle$`leq` : $\alpha \to$ `bool`$\rangle$: inheritance is not subtyping. Indeed, a method `leq` of an object of the former type expects to be applied to an object that has a method `getx`; this is not ensured by the latter type. However, `int_comparable` is an instance of ρ `#comparable`, which is by definition `rec` $\alpha.\langle$`leq` : $\alpha \to$ `bool`; $\rho\rangle$. Binary methods are correctly handled since the type of self is kept open while typing classes: adding the method `getx` to class `comparable` simply amounts to instantiating the row variable in the type of self, to `(getx : int; ..)`. Thus, the type of self in the subclass has a method `getx` and is still open.

As a test, the function `min` will return the minimum of any two objects whose type is an instance of type `#comparable`.

```
let min (x : #comparable) y =
  if x#leq y then x else y;;
value min : (#comparable as α) → α → α =
  ⟨fun⟩
```

This function can thus be applied to objects of type `int_comparable`.

```
let p = min (new int_comparable  7)
            (new int_comparable 11)
in (p, p#getx);;
- : int_comparable * int = ⟨obj⟩, 7
```

4. Proofs of type soundness theorems

Subject reduction is a straightforward combination of redex contraction (lemma 13) and context replacement (lemma 8).

Since we have multiple syntactic categories for expressions, contexts, and types, it is convenient to introduce the following meta-notations:

$$\check{a} ::= a \mid b \mid c \mid d \qquad \check{E} ::= E \mid F \mid E_c \mid F_d \qquad \check{\tau} ::= \tau \mid \varphi \mid \gamma$$

These meta-letters are used consistently. For instance, when writing $A \vdash \check{a} : \check{\tau}$, $(\check{a}, \check{\tau})$ means (a, τ), (b, φ), etc, but not (b, τ).

The following propositions are used several times in the proof.

Proposition 4 (Stability by substitution) *If $A \vdash \check{a} : \check{\tau}$, then for any substitution θ, $\theta(A) \vdash \check{a} : \theta(\check{\tau})$.*

Proposition 5 (Extension of environment) *If type environments A and B are identical on free variables of expression a and $A \vdash \check{a} : \check{\tau}$, then $B \vdash \check{a} : \check{\tau}$. If type environment B extends type environment A (that is $B \upharpoonright \mathrm{dom}\,(A)$ is A) and $A \vdash \check{a} : \check{\tau}$, then $B \vdash \check{a} : \check{\tau}$.*

We say that σ is an instance of σ' if any instance of σ is an instance of σ'. We say that type environment A is an instance of type environment A' if both type environments have the same domain and for any element h of their domain $A(h)$ is an instance of $A'(h)$.

Proposition 6 (Strengthening of context) *If type environment A is an instance of type environment B and $A \vdash a : \tau$, then $B \vdash a : \tau$.*

The following lemma somewhat simplifies the proofs.

Lemma 7 (Derivation simplification) *When proving that for all τ, $A_0 \vdash a_0 : \tau$ implies $A \vdash a : \tau$ (for some A_0, a_0, A and a), one can restrict oneself to the case where a derivation of $A_0 \vdash a_0 : \tau$ does not end with rule* Sub. *The general case follows.*

Proof. This is done by induction on the size of derivations. Let us assume that a derivation of $A_0 \vdash a_0 : \tau$ ends as

$$\frac{A_0 \vdash a_0 : \tau' \qquad \tau' \leq \tau}{A_0 \vdash a_0 : \tau}\ (\textsc{Sub})$$

By induction hypothesis, $A \vdash a : \tau'$. Hence

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

∎

We write $a_1 \subset a_2$ if for any environment A such that $A^\star = A$ and any type τ such that $A \vdash a_1 : \tau$, $A \vdash a_2 : \tau$. Likewise, we write $b_1 \subset b_2$ (resp. $c_1 \subset c_2$) if for any environments A and any class body type φ such that $A \vdash b_1 : \varphi$ (resp. any class type γ such that $A \vdash c_1 : \gamma$), then $A \vdash b_2 : \varphi$ (resp. $A \vdash c_2 : \gamma$). Subject reduction theorem can be restated as follows: if $a_1 \longrightarrow a_2$, then $a_1 \subset a_2$.

Lemma 8 (Context replacement) *For any context E, if $\check{a}_1 \subset \check{a}_2$ then $E[\check{a}_1] \subset E[\check{a}_2]$.*

Proof. The property can be proved independently for each arbitrary one-node context $\check{E}$. Then, the lemma follows by a trivial induction on the size of the context.

Let $\check{E}$ be a one-node context. Let A be a type environment and $\check{\tau}$ a type such that $A \vdash \check{E}[\check{a}_1] : \check{\tau}$ (1). We show that $A \vdash \check{E}[\check{a}_2] : \check{\tau}$. Using lemma 7, one can assume that a derivation of (1) does not end with rule Sub.

All cases are simple and similar. We show one case for example:

Case E is $\texttt{let}\ x = []\ \texttt{in}\ a$: A derivation of (1) ends as:

$$\frac{A \vdash a_1 : \tau' \qquad A + x : \mathtt{Gen}(\tau', A) \vdash a : \tau}{A \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a : \tau}\ (\textsc{Let})$$

By induction hypothesis applied to the first premise, $A \vdash a_2 : \tau'$. Hence $A \vdash \mathtt{let}\ x = a_2\ \mathtt{in}\ a : \tau$

∎

The following lemmas (9 thru 12) are used to simplify the proof of redex contraction.

Lemma 9 (Append) *Let A be a typing environment containing no* `super` *bindings. If $A \vdash b_1 : \varphi_1$, $A + (\varphi_1 \setminus \mathtt{method}) \vdash b_2 : \varphi_2$, and φ_1 and φ_2 are compatible (that is, $\varphi_1 \oplus \varphi_2$ is correct), then $A \vdash b_1 \mathbin{@} b_2 : \varphi_1 \oplus \varphi_2$.*

Proof. We actually prove a more general property. Let φ_0 be a sequence of `super` bindings. If $A + \varphi_0 \vdash b_1 : \varphi_1$, $A + (\varphi_1 \setminus \mathtt{method}) \vdash b_2 : \varphi_2$, and φ_1 and φ_2 are compatible (that is, $\varphi_1 \oplus \varphi_2$ is correct), then $A + \varphi_0 \vdash b_1 \mathbin{@} b_2 : \varphi_1 \oplus \varphi_2$.

This is easily proved by induction on b_1. ∎

Lemma 10 (Term replacement (variables)) *Let A be a type environment, $\check{a}$ and a' be term expressions, $\check{\tau}$ and τ' be type expressions. If $A^\star \vdash a' : \tau'$ (2) and $A + x : \mathtt{Gen}(\tau', A) \vdash \check{a} : \check{\tau}$* (3) *and bound variables of $\check{a}$ are not free in a', then $A \vdash \check{a}[a'/x] : \check{\tau}$ is provable* (4).

Proof. The proof is by induction on the structure of $\breve{a}$ (*i.e.* a, c, b and d). Using lemma 7, we can assume that a derivation of (3) does not end with rule SUB.

In each case, we consider a derivation of (3). By using a renaming substitution on (2) if necessary (proposition 4), we can assume that free variables of τ' that are not in $A^\star$ do not appear free in this derivation (5). We write A_x for $A + x : \texttt{Gen}(\tau', A^\star)$.

We only show the more complicated cases. Other cases are either similar or simple.

Case a is $\texttt{let}\ x_1 = a_1\ \texttt{in}\ a_2$: A derivation of (3) ends as:

$$\frac{(6)\ A_x \vdash a_1 : \tau_1 \qquad A_x + x_1 : \texttt{Gen}(\tau_1, A_x) \vdash a_2 : \tau\ (7)}{A_x \vdash \texttt{let}\ x_1 = a_1\ \texttt{in}\ a_2 : \tau}\ (\text{LET})$$

By induction hypothesis applied to (6), we get $A \vdash a_1[a'/x] : \tau_1$ (8).

If $x_1 = x$, (7) becomes $A + x : \texttt{Gen}(\tau_1, A_x) \vdash a_2 : \tau$. By strengthening of environment (proposition 6), we have $A + x : \texttt{Gen}(\tau_1, A) \vdash a_2 : \tau$ since A is a subsequence of A_x. We conclude by rule LET.

Otherwise, let A_1 be $A + x_1 : \texttt{Gen}(\tau_1, A)$. Re-ordering hypotheses in (7), we have $A + x_1 : \texttt{Gen}(\tau_1, A_x) + x : \texttt{Gen}(\tau', A) \vdash a_2 : \tau$. By strengthening of environment, we can replace A_x by A. Since free type variables of A_1 are the same as free type variables of A, we can replace A by A_1 in $\texttt{Gen}(\tau', A)$. Thus, we have $A_1 + x : \texttt{Gen}(\tau', A_1) \vdash a_2 : \tau$. On the other hand, since x_1 is not bound in a', and $A_1^\star$ extends $A^\star$, we deduce $A_1^\star \vdash a' : \tau'$ from (2) by extension of environment (proposition 5). Thus, we can apply the induction hypothesis with A_1 for A. We get $A_1 \vdash a_2[a'/x] : \tau$. Combining with (8) in a LET rule, we finally have $A \vdash (\texttt{let}\ x_1 = a_1\ \texttt{in}\ a_2)[a'/x] : \tau$.

Case a is $\texttt{fun}\ (x_1)\ a_2$: A derivation of (3) ends as:

$$\frac{A_x + x_1 : \tau_1 \vdash a_2 : \tau_2}{A_x \vdash \texttt{fun}\ (x_1)\ a_2 : \tau_1 \to \tau_2}\ (\text{FUN})$$

Let A_1 be $A + x_1 : \tau_1$. Re-ordering type environment of the premise, we have $A + x_1 : \tau_1 + x : \texttt{Gen}(\tau', A) \vdash a_2 : \tau_2$. By (5), the generalization $\texttt{Gen}(\tau', A)$ is equal to $\texttt{Gen}(\tau', A + x_1 : \tau_1)$, that is, $\texttt{Gen}(\tau', A_1)$. So, we have $A_1 + x : \texttt{Gen}(\tau', A_1) \vdash a_2 : \tau_2$. Since x_1 is not bound in a' and $A_1^\star$ extends $A^\star$, we deduce $A_1^\star \vdash a' : \tau'$ from (2). Thus, we can apply the induction hypothesis with A_1 for A. We get $A_1 \vdash a_2[a'/x] : \tau_2$. We conclude with rule FUN

Case a is $\langle b \rangle$: A derivation of (3) ends as:

$$\frac{A_x^\star + \texttt{self} : \tau_y \vdash b : \varphi \qquad \tau_y = \langle \texttt{method}\ (\varphi) \rangle}{A_x \vdash \langle b \rangle : \tau_y}\ (\text{OBJECT})$$

Let A_y be $A^\star + \texttt{self} : \tau_y$. Re-ordering type environment of the premise, we have $A^\star + \texttt{self} : \tau_y + x : \texttt{Gen}(\tau', A) \vdash b : \varphi$. We can replace $\texttt{Gen}(\tau', A)$ by $\texttt{Gen}(\tau', A^\star)$ by strengthening of environment. By (5), the generalization $\texttt{Gen}(\tau', A^\star)$ is equal to $\texttt{Gen}(\tau', A^\star + \texttt{self} : \tau_y)$, that is, $\texttt{Gen}(\tau', A_y)$. Thus, we have $A_y + x : \texttt{Gen}(\tau', A_y) \vdash b : \varphi$. Since $A_y^\star$ is just $A^\star$, we have $A_y^\star \vdash a' : \tau'$ (3). Thus, we can apply the induction hypothesis with A_y for A. We get $A_y \vdash b[a'/x] : \varphi$. We conclude with rule OBJECT.

∎

Lemma 11 (Term replacement (instance variables and $\texttt{self}$)) *Let A be an environment and $\breve{a}$ be either an expression a or a class expression c. Let w be an object body and φ be an object body type. We defines U as the restriction of $\mathrm{dom}\,(w)$ to fields. We write τ_y for $\langle \texttt{method}\ (\varphi) \rangle$. We assume that $A^\star$ is A, bound variables of $\breve{a}$ are not be free in $\langle w \rangle$ and $w(u)$, and the following three judgments hold:*

$$A + \texttt{self} : \tau_y \vdash w : \varphi, \qquad (A \vdash w(u) : \tau_u)^{u \in U}, \qquad A + \texttt{self} : \tau_y + (\varphi \setminus \texttt{method}) \vdash \breve{a} : \breve{\tau} (9).$$

Then, $A \vdash \breve{a}[\langle w \rangle/\texttt{self}][w(u)/u]^{u \in U}[\langle w\ @\ (\texttt{field}\ u = a_u{}^{u \in V}) \rangle / \{\langle u = a_u{}^{u \in V} \rangle\}]^{V \subset U} : \breve{\tau}$.

Proof. The proof is by induction on the structure of $\breve{a}$. For any expression a, we write a^+ for

$$a[\langle w \rangle/\texttt{self}][w(u)/u]^{u \in U}[\langle w\ @\ (\texttt{field}\ u = a_u{}^{u \in V}) \rangle / \{\langle u = a_u{}^{u \in V} \rangle\}]^{V \subset U}$$

Class expression c^+ is defined likewise. We write A_y for $A + \texttt{self} : \tau_y + (\varphi \setminus \texttt{method})$. Using lemma 7, we can assume that a derivation of (9) does not end with rule SUB.

We only show the more complicated cases. Other cases are easy.

Case a is self: Hypothesis (9) is $A + \texttt{self} : \tau_y + (\varphi \setminus \texttt{method}) \vdash \texttt{self} : \tau$. So, τ and τ_y are equal. On the other hand, a^+ is equal to $\langle w \rangle$. We conclude by rule OBJECT:

$$\frac{A + \texttt{self} : \tau \vdash w : \varphi \qquad \tau = \langle \texttt{method}\ (\varphi) \rangle}{A \vdash \langle w \rangle : \tau}\ (\textsc{Object})$$

Case a is $\{\langle u = a_u{}^{u \in V} \rangle\}$: A derivation of (9) ends as:

$$\frac{((10)\ \texttt{field}\, u : \tau_u \in A_y \qquad (11)\ \ A_y \vdash a_u : \tau_u)^{u \in V}}{A_y \vdash \{\langle u : a_u{}^{u \in V} \rangle\} : \tau_y}\ (\textsc{Override})$$

So, from (10), $\varphi \oplus \texttt{field}\ u : \tau_u{}^{u \in V} = \varphi$. By induction hypothesis applied to (11), we get $A \vdash a_u^+ : \tau_u$ (12). Hence $A \vdash (\texttt{field}\, u = a_u^+)^{u \in V} : (\texttt{field}\, u : \tau_u)^{u \in V}$. Then, the append lemma 9 applied to the hypothesis $A + \texttt{self} : \tau_y \vdash w : \varphi$ and the last judgment yields $A + \texttt{self} : \tau_y \vdash w\ @\ (\texttt{field}\, u = a_u^+)^{u \in V} : \varphi$. Hence the following derivation :

$$\frac{A + \texttt{self} : \tau_y \vdash w\ @\ (\texttt{field}\, u = a_u^+)^{u \in V} : \varphi \qquad \tau_y = \langle \texttt{method}\ (\varphi) \rangle}{A \vdash \langle w\ @\ (\texttt{field}\, u = a_u^+)^{u \in V} \rangle : \tau_y}\ (\textsc{Object})$$

■

Lemma 12 (Term replacement (super)) *If $A \vdash b_1 : \varphi_1$, $A + \texttt{super}: \varphi \vdash b_2 : \varphi_2$ and bound variables of b_2 are not free in b_1, then $A \vdash b_2' : \varphi_2$ where b_2' is $[a/s\#m]^{\texttt{method}\, m = a \in b_1}$,* i.e. *$b_2$ where all invocations of methods to super $s\#m$ have been replaced by the body a of the corresponding method m in b_1.*

Proof. The proof is similar to the one of lemma 10. It is in fact simpler, as super is not substituted across class and object boundaries, nor across instance variable definitions. ■

Lemma 13 (Redex contraction) *We write $\longrightarrow_\epsilon$ for a one-step reduction in an empty context. If $\breve{a}_1 \longrightarrow_\epsilon \breve{a}_2$ then $\breve{a}_1 \subset \breve{a}_2$.*

Proof. The proof is done independently for each redex. All cases are easy now that we have proven the right lemmas.

Let us assume $A \vdash a_1 : \tau$ (13) and A equals $A^\star$ (resp. $A \vdash b_1 : \varphi$ (14) for any A). We show that $A \vdash a_2 : \tau$ (15) (resp. $A \vdash b_2 : \varphi$) by cases on the redex a_1 (resp. b_1). Each case is shown independently. Using lemma 7, we can assume that a derivation of (13) does not end with rule SUB.

Case a_1 is $(\texttt{fun}\ (x)\ a)\ v$: A derivation of (13) ends either as:

$$\frac{\dfrac{\dfrac{A + x : \tau' \vdash a : \tau_0}{A \vdash \texttt{fun}\ (x)\ a : \tau' \to \tau_0}\ (\textsc{Fun}) \qquad \tau' \to \tau_0 \leq \tau_0' \to \tau}{A \vdash \texttt{fun}\ (x)\ a : \tau_0' \to \tau}\ (\textsc{Sub}) \qquad A \vdash v : \tau_0'}{A \vdash (\texttt{fun}\ (x)\ a)\ v : \tau}\ (\textsc{App})$$

or as:

$$\frac{\dfrac{(16)\ \ A + x : \tau' \vdash a : \tau}{A \vdash \texttt{fun}\ (x)\ a : \tau' \to \tau}\ (\textsc{Fun}) \qquad (17)\ \ A \vdash v : \tau'}{A \vdash (\texttt{fun}\ (x)\ a)\ v : \tau}\ (\textsc{App})$$

The end of the first derivation can be rewritten as:

$$\frac{\dfrac{\dfrac{A + x : \tau' \vdash a : \tau_0 \qquad \tau_0 \leq \tau}{(16)\ \ A + x : \tau' \vdash a : \tau}\ (\textsc{Sub})}{A \vdash \texttt{fun}\ (x)\ a : \tau' \to \tau}\ (\textsc{Fun}) \qquad \dfrac{A \vdash v : \tau_0' \qquad \tau_0' \leq \tau'}{(17)\ \ A \vdash v : \tau'}\ (\textsc{Sub})}{A \vdash (\texttt{fun}\ (x)\ a)\ v : \tau}\ (\textsc{App})$$

In both cases, the term replacement lemma 10 applied to (17) and (16) shows the conclusion.

Case c_1 is $(\texttt{fun}\ (x)\ c)\ v$: Similar to previous case.

Case a_1 is $\texttt{let}\ x = v\ \texttt{in}\ a$: A derivation of (13) ends as

$$\frac{(18)\ \ A \vdash v : \tau' \qquad (19)\ \ A + x : \texttt{Gen}(\tau', A) \vdash a : \tau}{A \vdash \texttt{let}\ x = v\ \texttt{in}\ a : \tau}\ (\textsc{Let})$$

The term replacement lemma 10 applied to (18) and (19) shows the conclusion.

Case a_1 is $\mathtt{class}\ z = v\ \mathtt{in}\ a$: Similar to previous case.

Case a_1 is $\mathtt{new}\ (\mathtt{struct}\ w\ \mathtt{end})$: A derivation of (13) ends as

$$\frac{\dfrac{A^\star + \mathtt{self} : \tau_y \vdash w : \varphi}{A \vdash \mathtt{struct}\ w\ \mathtt{end} : \mathtt{sig}\ (\tau_y)\ \varphi\ \mathtt{end}}\ (\textsc{Class-Body}) \qquad \tau_y = \langle \mathtt{method}\ (\varphi)\rangle}{(20)\ \ A \vdash \mathtt{new}\ (\mathtt{struct}\ w\ \mathtt{end}) : \tau_y}\ (\textsc{New})$$

Hence,

$$\frac{A^\star + \mathtt{self} : \tau_y \vdash w : \varphi \qquad \tau_y = \langle \mathtt{method}\ (\varphi)\rangle}{A \vdash \langle w \rangle : \tau_y}\ (\textsc{Object})$$

Case a_1 is $\langle w \rangle \# m$: We must remember that $A^\star$ is A. A derivation of (13) ends either as

$$\frac{\dfrac{\dfrac{A + \mathtt{self} : \tau_y \vdash w : \varphi \qquad \tau_y = \langle \mathtt{method}\ (\varphi)\rangle}{A \vdash \langle w \rangle : \tau_y}\ (\textsc{Object}) \qquad \tau_y \le {\tau_y}'}{A \vdash \langle w \rangle : {\tau_y}'}\ (\textsc{Sub}) \qquad {\tau_y}' = \langle m : \tau_k' ; \tau' \rangle}{A \vdash \langle w \rangle \# m : \tau_k'}\ (\textsc{Send})$$

or as

$$\frac{\dfrac{(21)\ \ A + \mathtt{self} : \tau_y \vdash w : \varphi \qquad (22)\ \ \tau_y = \langle \mathtt{method}\ (\varphi)\rangle}{A \vdash \langle w \rangle : \tau_y}\ (\textsc{Object}) \qquad (23)\ \ \tau_y = \langle m : \tau_k ; \tau \rangle}{A \vdash \langle w \rangle \# m : \tau_k}\ (\textsc{Send})$$

The end of the first derivation can be rewritten

$$\frac{\dfrac{\dfrac{A + \mathtt{self} : \tau_y \vdash w : \varphi \qquad \tau_y = \langle \mathtt{method}\ (\varphi)\rangle}{A \vdash \langle w \rangle : \tau_y}\ (\textsc{Object}) \qquad \tau_y = \langle m : \tau_k ; \tau \rangle}{A \vdash \langle w \rangle \# m : \tau_k}\ (\textsc{Send}) \qquad \tau_k \le \tau_k'}{A \vdash \langle w \rangle \# m : \tau_k'}\ (\textsc{Sub})$$

It has been seen at the beginning of the proof that rule Sub at the end of a derivation could be ignored. Thus, only the second case need to be considered.

The result is then proved using the term replacement lemma 11.

We first show that the hypotheses of lemma 11 are satisfied. As the fields of an object are typed in the same environment as the object, for $\mathtt{field}\ u : \tau_u \in \varphi$, $A \vdash v_u : \tau_u$ (24) where $\mathtt{field}\ u = v_u \in w$. From (22) and (23), $\mathtt{method}\ m : \tau_k \in \varphi$. Then, from (21), an easy induction on w using rules Then, Field, and Method yields:

$$A + \mathtt{self} : \tau_y + \varphi_1 \vdash w(m) : \tau_k \text{ for some } \varphi_1 \subset (\varphi \setminus \mathtt{method})$$

As A contains no $\mathtt{field}$ bindings, the environment can be extended to include $\varphi \setminus \mathtt{method}$:

$$(25)\ \ A + \mathtt{self} : \tau_y + (\varphi \setminus \mathtt{method}) \vdash w(m) : \tau_k$$

Finally, the term replacement lemma 11 applied to (21), (24), (25) yields

$$A \vdash w(m)[\langle w \rangle / \mathtt{self}][w(u)/u]^{u \in U}[\langle w \mathbin{@} (\mathtt{field}\ u = {a_u}^{u \in V})\rangle / \{\langle u = {a_u}^{u \in V}\rangle\}]^{V \subset U} : \tau_k$$

Case b_1 is $\mathtt{inherit}\ (\mathtt{struct}\ w\ \mathtt{end})\ \mathtt{as}\ s\ ; b$: A derivation of (14) ends as

$$\frac{\begin{matrix}\vdots \\ A \vdash \mathtt{inherit}\ (\mathtt{struct}\ w\ \mathtt{end})\ \mathtt{as}\ s : \varphi\end{matrix} \qquad (26)\ \ A + (\varphi \setminus \mathtt{method}) \vdash b : \varphi_2}{A \vdash \mathtt{inherit}\ (\mathtt{struct}\ w\ \mathtt{end})\ \mathtt{as}\ s\ ; b : \varphi_1 \oplus \varphi_2}\ (\textsc{Then})$$

where $\varphi = \varphi_1 + (\mathtt{super}\ s : \varphi_1)$, continued by

$$\frac{(27)\ \ A \vdash \mathtt{self} : \tau_y \qquad \dfrac{(28)\ \ A^\star + \mathtt{self} : \tau_y \vdash w : \varphi_1}{A \vdash \mathtt{struct}\ w\ \mathtt{end} : \mathtt{sig}\ (\tau_y)\ \varphi_1\ \mathtt{end}}\ (\textsc{Class-Body})}{A \vdash \mathtt{inherit}\ (\mathtt{struct}\ w\ \mathtt{end})\ \mathtt{as}\ s : \varphi_1 + (\mathtt{super}\ s : \varphi_1)}\ (\textsc{Inherit})$$

According to (27), $\texttt{self} : \tau_y \in A$. Judgment (28) can thus be rewritten $A \vdash w : \varphi_1$ (29).

Applying the term replacement lemma 12 on $A + (\varphi_1 \setminus \texttt{method}) \vdash w : \varphi_1$ (the environment has been extended) and (26) yields $A + (\varphi_1 \setminus \texttt{method}) \vdash b[a/s\#m]^{\texttt{method}m=a\in w} : \varphi_2$. Then, the append lemma applied on (29) and this last judgment gives the result:

$$A \vdash w \mathbin{@} b[a/s\#m]^{\texttt{method}m=a\in w} : \varphi_1 \oplus \varphi_2$$

Case b_1 is $\texttt{field}\ u = v\ ; b$: A derivation of (14) ends as

$$\dfrac{\dfrac{A^\star \vdash v : \tau}{A \vdash \texttt{field}\ u = v : (\texttt{field}\ u : \tau)}\ (\textsc{Field}) \qquad (30)\ \ A + (\texttt{field}\ u : \tau) \vdash w : \varphi}{A \vdash \texttt{field}\ u = v\ ; w : \varphi \oplus (\texttt{field}\ u : \tau)}\ (\textsc{Then})$$

From (30), since $u \in dom\,(w)$ and fields appear before methods in w, an easy induction shows that $A \vdash w : \varphi$. Indeed, fields are typed in environment $A^\star$, and methods are typed in an environment in which $(\texttt{field}\ u : \tau)$ has been added anyway after the typing of the field u appearing in w.

Case b_1 is $\texttt{method}\ m = a\ ; b$: A derivation of (14) ends as

$$\dfrac{\dfrac{A \vdash \texttt{self} : \langle m : \tau; \tau' \rangle \qquad A \vdash a : \tau}{A \vdash \texttt{method}\ m = a : (\texttt{method}\ m : \tau)}\ (\textsc{Method}) \qquad (31)\ \ A \vdash w : \varphi}{A \vdash \texttt{method}\ m = a\ ; w : (\texttt{method}\ m : \tau) \oplus \varphi}\ (\textsc{Then})$$

Since $m \in dom\,(w)$, $m \in dom\,(\varphi)$, then φ and $(\texttt{method}\ m : \tau) \oplus \varphi$ are equal. Therefore, judgment (31) can be rewritten $A \vdash w : (\texttt{method}\ m : \tau) \oplus \varphi$.

Case a_1 is $(v : \tau <: \tau')$: A derivation of (13) ends as

$$\dfrac{A \vdash v : \theta(\tau) \qquad \tau \leq \tau'}{A \vdash (v : \tau <: \tau') : \theta(\tau')}\ (\textsc{Coerce})$$

Hence,

$$\dfrac{A \vdash v : \theta(\tau) \qquad \theta(\tau) \leq \theta(\tau')}{A \vdash v : \theta(\tau')}\ (\textsc{Sub})$$

■

The normal-form theorem is proved by structural induction on values, using the following lemma.

Lemma 14 *Let v be a value. We assume $\emptyset \vdash v : \tau$ (32).*

- If τ is a functional type, then v is a function.
- If τ is an object type, then v is an object.

Let v_c be a value. We assume $\emptyset \vdash v_c : \gamma$.

- If γ is a functional type, then v is a function.
- Otherwise, v is an object.

Proof. We prove that if v is a function, then τ is a functional type and that if v is an object, then τ is an object type. Then, since a value is either a function or an object and functional types and object types are incompatible, this proves the lemma.

We can ignore rule Sub at the end of a derivation, as it does not change the shape of a type.

Case a is $\texttt{fun}\ (x)\ a_1$: A derivation of (32) ends as

$$\dfrac{A + x : \tau_1 \vdash a_1 : \tau_2}{A \vdash \texttt{fun}\ (x)\ a_1 : \tau_1 \to \tau_2}\ (\textsc{Fun})$$

So, τ is $\tau_1 \to \tau_2$.

Case a is $\langle w \rangle$: A derivation of (32) ends as

$$\dfrac{A^\star + \texttt{self} : \tau_y \vdash w : \varphi \qquad \tau_y = \langle \texttt{method}\ (\varphi) \rangle}{A \vdash \langle w \rangle : \tau_y}\ (\textsc{Object})$$

So, τ is $\langle \texttt{method}\ (\varphi) \rangle$.

The proof is similar for class values.

■

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.)

Proof. The proof is by structural induction simultaneously on expressions a and class bodies b. Let us assume $\emptyset \vdash a : \tau$ (33) (resp. $\emptyset \vdash c : \gamma$ (34), $A \vdash b : \varphi$ (35) or $A \vdash d : \varphi$, where A contains only `field` and `method` bindings), and that a (resp. c, b or d) cannot be reduced.

Case a is x: This expression cannot be typed in the empty environment.

Case a is $a_1\ a_2$: It is not possible. A derivation of (33) shows that there exists a type τ_1 such that $\emptyset \vdash a_1 : \tau_1 \rightarrow \tau$. The induction hypothesis applied to expression a_1 shows that it is a value. Since it has a functional type, it must be a function $\texttt{fun}\ (x)\ a_0$. But then expression a could be reduced.

Case a is $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$: It is not possible. The induction hypothesis applied to expression a_1 shows that it is a value. But then expression a could be reduced.

Case a is $a_1\#m$ or $\texttt{class}\ z = c\ \texttt{in}\ a_1$: Similar to previous cases.

Case a is $\texttt{fun}\ (x)\ a_1$: By definition, expression a is a value.

Case a is $s\#m$: It is not possible : expression $s\#m$ is not typable in the empty environment.

Case a is `self` or u or $\{\langle u = a_u{}^{u \in V}\rangle\}$: Same as previous case.

Case a is $(a_1 : \tau <: \tau')$: It is not possible: a can be reduced.

Case a is $\langle b \rangle$: The induction hypothesis shows that object body b is a value. Then, expression a is also a value.

Case a is $\texttt{new}\ c$: It is not possible. A derivation of (33) shows that $\emptyset \vdash c : \texttt{sig}\ (\tau_y)\ \varphi\ \texttt{end}$. The induction hypothesis applied to c shows that it is a value. According to its type, it is a structure. But then a can be reduced

Case c is z: This expression is not typable in the empty environment.

Case c is $c_1\ a$: It is not possible. A derivation of (34) shows that there exists a type τ such that $\emptyset \vdash c_1 : \tau \rightarrow \gamma$. The induction hypothesis applied to expression c_1 shows that it is a class value. Since it has a functional type, it must be a function $\texttt{fun}\ (x)\ c_0$. But then expression c could be reduced.

Case c is $\texttt{fun}\ (x)\ c_1$: By definition, expression c is a value.

Case c is $\texttt{struct}\ b\ \texttt{end}$: The induction hypothesis shows that class body b is a value. Then, expression c is also a value.

Case b is $d; b_1$: The induction hypothesis shows that object component d and object body b_1 are in normal forms. d is thus a field or method definition, and it is not overridden by b_1 (otherwise, b could be reduced.)

Case b is $\emptyset$: By definition, object body b is a value.

Case d is $\texttt{inherit}\ c\ \texttt{as}\ s$: It is not possible. A derivation of (35) ends as:

$$\frac{A \vdash \texttt{self} : \tau_y \qquad (36)\ \ A \vdash c : \texttt{sig}\ (\tau_y)\ \varphi_1\ \texttt{end}}{A \vdash \texttt{inherit}\ c\ \texttt{as}\ s : \varphi_1 + (\texttt{super}\ s : \varphi_1)}\ (\textsc{Inherit})$$

The induction hypothesis applied to c shows that it is a class value. According to its type, it is of the form $\texttt{struct}\ w\ \texttt{end}$. But then, the inheritance clause could be reduced.

Case d is $\texttt{method}\ m = a$: By definition, expression d is in normal form.

Case d is $\texttt{field}\ u = a$: If $A \vdash d : \texttt{field}\ u : \tau$, then $\emptyset \vdash a : \tau$, as A contains only `field` and `method` bindings. By induction hypothesis, expression a is in normal form. Then, so is object component d.

∎