To appear in TACS'94

Programming Objects with ML-ART
An extension to ML with Abstract and Record Types

Didier Rémy

INRIA-Rocquencourt, BP 105, F-78153 Le Chesnay Cedex.

Abstract. Class-based objects can be programmed directly and efficiently in a simple extension to ML. The representation of objects, based on abstract and record types, allows all usual operations such as multiple inheritance, object returning capability, and message transmission to themselves as well as to their super classes. There is, however, no implicit coercion from objects to corresponding ones of super-classes. A simpler representation of objects without recursion on values is also described. The underlying language extends ML with recursive types, existential and universal types, and mutable extensible records. The language ML-ART is given with a call-by-value semantics for which type soundness is proved.

Introduction

An important motivator for type-checking extensible records is their application to object encoding. Initiated by Cardelli in 1984 [Car84], continued by Wand [Wan87], and then many others, record type-checking has produced several satisfactory solutions for higher order languages [CM89, HP90] and for ML [JM88, OB88, Oho90, Rém93b]. Object encoding, based on record calculi, has revealed severe difficulties, mainly due to over-reliance on recursive values. Consequently, the tendency has been to design languages with objects as primitive operations [Bru93, AC94, Hen91a, MHF93], rather than encodings, to achieve important simplification of type-theoretical models.

Pierce and Turner produced convincing evidence that object-oriented programming can be treated as a matter of programming style, at least from a theoretical point of view [PT93]. However, the use of $F^{\omega}_{<:}$ as the base language supports the idea that encodings involve complex type theories, and the demonstration does not always apply to the ML programmer. The need to write many coercions, due to the use of explicit types and to the absence of record extension, makes it obvious that large-scale object-oriented applications cannot be programmed directly in $F^{\omega}_{<:}$. Finally, the encoding is created in a call-by-name language, which results in a duplication of too many structures. A recent version of the encoding in a call-by-value language [Pie93] still contains inherent inefficiencies. At least a large amount of syntactic sugar must be provided to program objects in $F^{\omega}_{<:}$.

We concur with the claim that object-oriented programming is essentially a matter of style. Consequently, we do not address it in this paper. Our main goal is to demonstrate that objects can be programmed in a small extension to ML. Therefore, we repeat Pierce's method utilizing, however, a basic language derived from ML. This results in a quite elegant and still flexible class-based object-oriented programming style, almost as concise as if objects were primitive. No syntactic sugar is required. This approach enables programming capabilities such as multiple inheritance, object returning ability and message transmission to self as well as super class. We recognize that implicit coercion of objects to their counterparts in super classes is not possible. This is a serious restriction of our approach to objects.

As in [PT93], we consider objects as abstract data structures, but our encoding differs in two essential ways. First, we can take advantage of record extension to implement inheritance in a simpler way which avoids successive coercions and treats classes as "first class citizens". Ignoring implicit class coercions enables us to move the recursion on "self" from method vector creation to method application, converting objects to non recursive values.

The second interest of this paper is the language ML-ART utilized for programming objects; it extends core ML with several orthogonal features. None of these is really new, however, the combination is original. We give a complete definition of ML-ART and verify type soundness, but we omit type inference.

The most important feature of ML-ART is extensible records. We choose those described in [Rém93b], although other choices are permissible, provided they implement polymorphic access and polymorphic extension. Polymorphic access refers to the ability to define a function that reads the same field of many records, with different domains. This is the key operation to message passing. Record extension is the operation that creates new records from older ones, via addition of new fields. It is said to be polymorphic if it can operate on records with different domains. Record extension is used to program single inheritance [Wan87] or even multiple inheritance reusing the trick that provides record concatenation through record extension only [Rém93c].

The language is also enriched with recursive types. Record types and recursive types are sufficient to program objects with value abstraction, modulo serious inefficiencies and difficulties with the compilation of recursive values. Thus, we choose to extend the language with existential types, as described by Laüfer and Odersky [LO92], and use type abstraction to conceal the internal state of objects. This necessitates replacement of record types with more expressive projective types [Rém92b]. Finally, existential types introduce scope borders which can only be crossed using universal types in a dual way.

The paper is organized as follows. In section 1, we compare encodings of objects with type and value abstraction. In section 2 we informally introduce the language ML-ART. We motivate and introduce its components, one by one. A formal presentation is supplied in the appendices. In section 3, we show how objects can be programmed in ML-ART. In the final section, we discuss and conclude the experience.

1 Objects cannot be programmed with value abstraction

Abstractly, objects are first class structures which can only be reached by messages. Moreover, a message may be sent to many objects that do not necessarily accept the same set of messages. In a very simplistic view, messages are labels, objects are records $\{a_1 = f_1; \ldots a_n = f_n\}$ whose domains are the set of messages that the objects may receive, and each value f_i is a function that describes the reaction of the object when it receives message a_i.

In practice, however, objects belong to classes. A class is a collection of objects that answer exactly the same set of messages and whose functions f_i's can be written $m_i\ r$ where r only depends on the object. The function m_i is called the method, and r is called the state. An object can thus be seen as a record of closures $\{a_1 = \langle r, m_1\rangle; \ldots a_n = \langle r, m_n\rangle\}$ or equivalently $\langle r, \{a_1 = m_1; \ldots a_n = m_n\}\rangle$, which we write $\langle r, \vec{m}\rangle$ and $\overrightarrow{\langle r, m\rangle}$, respectively, for short. This equivalence has been studied more formally in [HP92].

Yet, the two representations are quite different in practice. The representation $\langle r, \vec{m}\rangle$ enables sharing of the record of methods $\vec{m}$ while the other does not. When objects are primitive in the language, they can be treated as if they were records of closures, and can still be efficiently compiled by pairing the state with a record of methods, provided this is done consistently everywhere. Conversely, objects should rather be represented by a closure $\langle r, \vec{m}\rangle$ when they are derived constructs or directly programmed in the language.

The advantage of objects as records of closures is that states of objects are hidden in the partial application of m_i's to r. This guarantees that they cannot be directly accessed, except via messages. In the other view, objects expose their state. Access to the state can easily be protected by tagging objects with a non-accessible constructor. However, this is insufficient in a typed language, since the types will reveal the internal state of objects. Although this does no violate security, it creates incompatible sets of objects with identical interfaces, but different internal states. For instance, bidimensional points implemented with polar and cartesian coordinates could not be interchanged. More generally, different summands of the same data structure (for instance, nil and cons objects) cannot be viewed as two implementations of the same specification (the list interface).

As shown in [PT93], the state of objects can be nicely hidden using type abstraction. Higher-order type operators also play a key-role. When objects are viewed as pairs $(r, \vec{m})$ of an internal state r and a vector of methods $\vec{m}$, the type of r, say τ_r, must be hidden, but still allow r to be passed to any field of $\vec{m}$. Thus, each method m_i must have type $\tau_r \rightarrow \tau_i$, where the codomain of the methods, described by $\vec{\tau}_i$, differs on most fields. There must be a uniform way of describing *a pair whose first component has some type τ_r and whose second component is a record of functions of common domain τ_r, and different codomains.* In [PT93] , the language $F^{\omega}_{<:}$ provided both existential types and powerful type operators. Here, the base language is ML. It is immediately extended with extensible records [Rém93b] and a form of existential [LO92]. Then, projective types [Rém92b] are added as a palliation to the lack of type operators.

2 An informal introduction to ML-ART

We have shown evidence that before any attempt to program objects, the language ML must be extended with several features. Each extension is quite simple and none of them is really new; either they have been described somewhere else, or already implemented in some version of ML. Their combination provides just enough power to program objects in a flexible and elegant way. The two main extensions are polymorphic records and existential types. Recursive types are also added and record types are enriched to projective types. Finally, a simple form of universal types are added, imitating existential type, so that messages can be first class citizens.

In this section we briefly and informally describe the language ML-ART. A complete definition of the language is given in the appendices.

2.1 The core language

The core language is ML, with a call-by-value semantics. Programs are given by the following grammar:

$$a ::= x \mid c \mid \mathsf{fun}\ x \rightarrow a \mid a_1\ a_2 \mid \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 \mid (\mathrm{ref}\, a_1) \mid (!a_1) \mid (a_1 := a_2)$$

and are taken modulo renaming of bound variables. The conditionals and pairs may be provided as syntactic sugar and new constants (ranged over by c). For convenience, we also use simultaneous "let" bindings with the construction $\mathsf{let}\ x_1 = a_1\ \mathsf{and}\ \ldots\ x_n = a_n\ \mathsf{in}\ a_0$ that can be expanded into cascades of lets after renaming of bound variables.

Typing rules are usual Damas-Milner ones, restricting however, polymorphism to values for safety of mutable operations [Wri93].

2.2 Extensible records

Monomorphic records, such as those of Sml [HMT91] or Caml-Light [LM92], are not sufficient to program objects. The basic operation on objects is message passing. It is commonly implemented as an access to the appropriate method, through a record carried by the object itself. The same message often needs to be passed to objects of different classes, i.e. objects that can receive different sets of messages. Thus, access to the record of methods must be polymorphic.

Record extension is not absolutely required for simulating objects. For instance, in [PT93], classes are defined at top-level. Consequently, when one class inherits another, all methods of the super class are known and can be explicitly copied into the new record of methods. However, writing all coercion functions quickly becomes a burden and some syntactic sugar is required to automatically generate them. Non-polymorphic record extension can be useful to avoid syntactic sugar, but classes cannot yet be first-class citizens [Hen91b]. Polymorphic extension enables both multiple inheritance and classes as first-class citizens.

The extensible records are those[1] described in [Rém93b]. We assume that a denumerable collection of labels $\mathcal{L}$ is given. Instead of introducing new syntax for records, we extend the set of constants with the empty record $\{\}$ and two families of primitives $(\_.\ell)$ and $(\_ \parallel \{\ell = \_\})$ for all labels ℓ. These implement respectively the access to field ℓ and extension on field ℓ. For convenience, we also write $(a \parallel \{\ell_1 = a_1; \ldots \ell_n = a_n\})$ as a short hand for $(\ldots(a \parallel \{\ell_1 = a_1\})\ldots \parallel \{\ell_n = a_n\})$ and we omit $a \parallel$ when a is an empty record. As in Sml, we use the abusive but very convenient convention that $(a \parallel \{x\})$ stands for $(a \parallel \{\ell_x = x\})$, where ℓ_x is the label that has the same name as variable x.

ML types are enriched with record types:

$$\tau ::= \ldots \mid \{\tau\} \mid \tau.\tau \mid \mathsf{abs} \mid \mathsf{pre} \mid (\ell : \tau; \tau) \quad \ell \in \mathcal{L}$$

The formation of record types is restricted by sorts. Types are also taken modulo equations in order to re-arrange fields. For instance, the types $\{\ell_1 : \tau_1; (\ell_2 : \tau_2; \tau_3)\}$ and $\{\ell_2 : \tau_2; (\ell_1 : \tau_1; \tau_3)\}$ are equal. The types that are shown to the user can be put in the canonical form $\{\ell_1 : \tau_1; \ldots \ell_n : \tau_n; \tau_0\}$ where:

- each τ_i is either a variable or of the form $\tau_i'.\tau_i''$.
- τ_i' is either a variable or one of the two symbols pre and abs; it manifests the presence of the corresponding field,
- τ_i'' provides the type of the corresponding field if present,
- τ_0 is called the *template*; it identifies the presence and the types of all fields but $\ell_1, \ldots \ell_n$. To accomplish this, the template can be replaced by $(\ell_{n+1} : \tau_0'; \tau_0'')$ at any time, but consistently in all record types, where τ_0' and τ_0'' are two copies of τ_0. This operation is called expansion.
- variables that appear in the template, called template variables, may only occur in another template that is preceded by the same set of fields. In particular, they cannot occur outside of a template and, consequently, outside of a record type.

For instance, the record $\{\mathsf{a} = 2; \mathsf{b} = \mathsf{true}\}$ has type

$$\{\mathsf{a} : \mathsf{pre.true};\ \mathsf{b} : \mathsf{pre.bool};\ \mathsf{abs.}\ \mathsf{'a}\},$$

which says that fields a and b are present with types int and true, and all other fields are absent with any type; by expansion, the record also has type:

$$\{\mathsf{a} : \mathsf{pre.true};\ \mathsf{b} : \mathsf{pre.bool};\ \mathsf{c} : \mathsf{abs.}\ \mathsf{'b};\ \mathsf{abs.}\ \mathsf{'c}\}.$$

See [Rém93b] or the appendices for a detailed treatment of sorts and record types.

[1] In fact, in [Rém93b] two variants of record types are described, both in section 3.3. Here, we use the second one, but with the weaker type assumptions of the first one.

There are no special typing rules for records; the primitives simply come with the following principal types:

$$\begin{aligned}
\{\} &: \forall\, \alpha.\, \{\mathsf{abs}.\alpha\} \\
(_.l) &: \forall\, \alpha_0, \alpha_1.\, \{l : \mathsf{pre}.\alpha_0; \alpha_1\} \rightarrow \alpha_0 \\
(_ \mid\mid \{\ell = _\}) &: \forall\, \alpha_0, \alpha_1, \alpha_2.\, \{l : \alpha_0; \alpha_1\} \rightarrow \alpha_2 \rightarrow \{l : \mathsf{pre}.\alpha_2; \alpha_1\}
\end{aligned}$$

2.3 Projective types

As concluded in section 1, it should be possible to write the type of records of functions that have the same domain but different codomains. Simple record types enable types in templates. For instance, the type $\{\alpha_2.\alpha_0 \rightarrow \alpha_1\}$ describes records whose fields always carry functions. It is impossible, however, to write such type as $\alpha_0 \times \{\alpha_2.\alpha_0 \rightarrow \alpha_1\}$ since α_0 is used both in templates and outside of them.

Projective types solve this problem by introducing a new symbol $[\_]$ used to coerce standard types to templates. It prevents the type τ from being copied when $[\tau]$ is expanded. The above type can be written $\alpha_0 \times \{\alpha_2.[\alpha_0] \rightarrow \alpha_1\}$. By expansion, a value of this type also has type $\alpha_0 \times \{\ell : \alpha'_2.\alpha_0 \rightarrow \alpha'_1;\ \alpha''_2.[\alpha_0] \rightarrow \alpha''_1\}$. Thus $\tau_0 \times \{\alpha.[\tau_0] \rightarrow \tau_1\}$ is the type of pairs composed of a value of type τ_0 and a record whose fields are functions of domains τ_0 and of codomains described by τ_1, i.e. the type of objects with exposed state τ_0.

Projective types are described in more details in the appendix B. They are fully formalized in [Rém93a] and also introduced more intuitively in [Rém92b]. In [Rém92b] projective types are used to type more powerful primitives. In ML-ART they essentially define more precise types such as the type of objects.

2.4 Recursive types

Recursive types are only provided through data-type declarations in ML. Since objects must be able to send messages to themselves, either the objects or the functions that send messages to objects have recursive types. In order to avoid type declaration of objects in ML-ART, we allow implicit recursive types. Adding implicit recursive types to ML is quite easy since type inference reduces to first order unification, and there are well-known unification algorithms for recursive types [MM82, Hue76].

Equality for recursive types, defined in appendix B, is standard [AC91], but recursive types need to be considered more carefully in the presence of equations. To understand recursive types intuitively, one should think of them as regular trees. In order to define them we add the syntax $\mathsf{rec}\ \alpha\ \mathsf{in}\ \tau$, where τ is neither a variable nor another $(\mathsf{rec}\ \_\ \mathsf{in}\ \_)$ type.

With recursive types, the call-by-value fix-point combinator

```
let Y  F = (fun f x → F (f f) x) (fun f x → F (f f) x);;
```

can be defined. Recursive definition of functions $\mathsf{let\ rec}\, f = a_1\ \mathsf{in}\ a_2$ is allowed as syntactic sugar for $\mathsf{let}\ f = \mathsf{Y}\ (\mathsf{fun}\ f \rightarrow a_1)\ \mathsf{in}\ a_2$. Recursively let-bound variables become lambda-bound in the expanded form, which provides the usual monomorphic typing rules for recursion. The type of Y forces a_1 to have a functional type. Consequently, it does not allow the construction of non-functional recursive values.

2.5 Existential types

Existential types are the basic tool for defining objects with type abstraction. An extension of ML with existential types has been proposed by K. Läufer and M. Odersky in [LO92]. In ML-ART we take a slightly simpler version of existential types by separating them from variant types. We briefly and informally introduce them here. They are formalized in the appendix and more thoroughly described in [LO92, Läu92]. Existential types are introduced by type declarations similar to data-type declarations:

$$\mathsf{type}\ \ \kappa(\tau_0)\ =\ K\ \ \mathsf{of\ Exist}\ (\vec{\alpha})\ \tau_1$$

However such a declaration differs in two ways:

– Some variables of the body τ_1 of the definition may not occur inside the argument τ_0 of the new type symbol κ. Those variables must be listed in $\vec{\alpha}$.
– There is only one summand.

The intuition behind such a type definition lies in the fact that the type $\kappa(\tau_0)$ abbreviates the higher order existential type $\exists\,\vec{\alpha}.\,\tau_1$. The term τ_0 could be restricted to be a tuple of variables, as in ML. The more general form avoids complicated sort constraints. It was already used in the language LCS [Ber93].

For example,

```
type 'a freeze = Freeze of Exist ('u) ('u → 'a) * ('u * 'u);;
```

The syntax of the language is extended with existential introduction and elimination constructs

$$a ::= \ldots \mid K\ a \mid \mathsf{let}\ K\ x = a_1\ \mathsf{in}\ a_2$$

They tell when to pack values as abstract values and when to open them. An abstract value is created in the same maner as a value of a concrete data structures:

```
let ice = Freeze (succ, (0,1));;
```

Roughtly speaking, when opening an existential value a_1 of type $\kappa(\tau_0)$ as $K\ x$ in a_2, variable x is assigned type τ_1 where variables $\vec{\alpha}$ have been replaced by new type symbols $\vec{\Omega}$ that must not occur in the type of a_2 (see typing rules in the appendix). For instance, one can write

```
let (Freeze (f,(x,y))) = ice in f x;;
```

since (f,(x,y)) is given type (U → int) * (U * U) where U is a new symbol and f “@ x can be typed with int. However, the program

```
let apply g ice = let (Freeze (f,p)) = ice in f (g p) in apply fst ice;;
```

fails to type, since type symbol U is transmitted to the type of apply through variable g.

2.6 Universal types

Opening an abstract type introduces new type symbols with restricted scope. These symbols are quickly propagated by unification outside of their scope. The above example is not type-correct since the argument g is monomorphic and captures the abstract type symbol U in the type of its argument. A solution is to pass g polymorphically and then to take an instance inside the scope of the de-structuring let expression.

Universal types are simpler than existential types since they do not require the introduction of new type symbols. They are defined as the existential types with the same restrictions and manipulated with the same syntactic construct:

$$\mathsf{type}\quad \kappa(\tau_0) \;=\; K \;\mathsf{of}\; \mathsf{All}\; (\vec{\alpha})\; \tau_1$$

However, the creation of a universal value $K\ a_1$ may fail to type if the type of the expression a_1 is not as general as the expected type scheme $\forall\,(\vec{\alpha}).\,\tau_1$. When opening a universal value a_1 of type $\kappa(\tau_0)$ as $K\ x$ in a_2, variable x is assigned type τ_1 where variables $\vec{\alpha}$ are replaced by fresh variables (see typing rules in the appendix). For instance, we may define:

```
type projection = Projection of All ('a, 'b) 'a * 'b → 'a;;
```

and type the combined example:

```
let apply G ice =
 let (Freeze (f,x)) = ice in let (Projection g) = G in  f (g x) in
apply (Projection fst) ice;;
| - : int = 1
```

Here, the abstract type symbol U that appears in types of f and x is communicated to the type 'a1 * 'b1 → 'a1 of g which is only an instance of the type scheme projection of G that abbreviates ∀ ('a, 'b). 'a * 'b → 'a. Therefore, U does not appear in the type of apply any longer.

3 Objects and Inheritance

With the rich type system of ML-ART we can now attempt to define the types of objects. In this section we show how to program objects and inheritance with type abstraction.

All examples are run in a prototype implementation of ML-ART, that has been implemented from the Caml-Light system [LM92]. The language ML-ART is strongly typed and provides type inference. However, objects have anonymous, long, and often recursive types that describe all methods that the object can receive. Thus, we usually do not show the inferred types of programs in order to emphasize object and inheritance encodings rather than typechecking details. This is quite in the spirit of ML where type information is optional and is mainly used for documentation or in module interfaces. Except when trying top-level examples, or debugging, the user does not often wish to see the inferred types of his programs in a batch compiler. When printed, the output of top-level evaluation is indicated with a marginal "❚" sign. Counter-examples, which fail to type, are marked with barbed wires "✗" and the typing failure is explained in English rather than in machine-spoken language.

3.1 A first attempt at programming objects

In section 2.3, we have seen that $\tau_0 * \{\alpha.[\tau_0] \rightarrow \tau_1\}$ is the type of objects with exposed state τ_0. Thus we define the type of objects by abstracting the state:

```
type ({'presence.'methods}) object =
    Object of Exist ('R) 'R * {'presence. ['R] → 'methods};;
```

An object point could be defined as follows. We first define its representation, then its method vector, last we combine the two:

```
let pointR v = {x = v};;

let pointM =
  let getx R = R.x in
  let print R = print_int (getx R) in
  {getx; print};;

let point v = Object (pointR v, pointM);;
```

The print method of points explicitly uses the method getx that has been defined simultaneously. If the getx method is later redefined in some other kind of points, for instance,

```
let anti_pointM = let getx R x = −R.x in pointM || {getx};;
```

then the print method of anti-points still uses the method getx of pointM. The correct definition of print must take getx from the methods of the object itself rather than from a previously defined record of methods. The well-known solution is to pass the self methods M as a parameter to the definition of pointM:

```
let pointM M =
  let getx R = R.x
  and print R = print_int (M.getx R)
  in {getx; print};;
```

However, the creation of points has to build the record of methods recursively as follows:

```
✗ let point v = let rec M = pointM M in Object (pointR v, M);;
```

This kind of recursive definition is not allowed. The example could be compiled correctly, but the general case is unsafe. It should be verified that the expression pointM M does not access fields of M before they are filled. This is obvious here because pointM is a record of values, but the general case requires some non trivial analysis.

3.2 Objects without recursive values

There are several ways to realize recursion on some non-functional values. They are discussed in section 3.7. However, we can take advantage of the fact that there is no subtyping on objects to implement a simpler solution. Anyway, all examples of this section can be adapted to the previous representation of objects if one prefers to keep objects as recursive records.

Going from objects with value abstraction to objects with type abstraction, we have moved the abstraction on state from outside the record of methods into each method. Similarly, we can move abstraction on M into methods themselves. For instance, the method print can be defined as:

```
let print (R,M) = print_int (M.getx (R,M));;
```

Each method should now take the record of methods together with the state as argument. This forces methods to have recursive types, and unsurprisingly, the type of objects must be redefined to:

```
type ({'presence.'methods}) object =
  Object of Exist ('R) rec 'RM in 'R * {'presence. ['RM] → 'methods};;
```

The new implementation of pointM is

```
let pointM =
  let getx (R,M) = R.x
  and print (R,M) = print_int (M.getx (R,M)) in
  {getx; print};;

let point v = Object (pointR v, pointM);;
```

Then anti_pointM can be defined by re-using methods from pointM and have the expected behavior.

The simplest way to send objects messages is to define a send function for each message:

```
let send_getx P = let (Object (R, M)) = P in M.getx (R,M);;
```

Another option is to view messages as field extractors,

```
let getx = fun z → z.getx;;
```

and define a unique send function. Unfortunately, the following function fails to type:

```
let send extractor P =
  let (Object (R, M)) = P in extractor M (R,M);;
```

The abstract type of (R,M) propagates to the type send through variable extractor, and the scope of the abstraction is violated. As in the example of section 2.6, the solution is to make extractors polymorphic on the representation, so that the abstract representation is not exported through the extractor.

```
type ({'presence.'methods}, 'a) extractor = Extractor of
  All ('R,'M)
```

```
    {'presence. (['R] * 'M) → 'methods} →
    ('R * {'presence. (['R] * 'M) → 'methods}) → 'a;;

let getx = Extractor (fun z → z.getx);;
let print = Extractor (fun z → z.print);;

let send extractor P =
  let (Object (R,M)) = P in let (Extractor x) = extractor in x M (R,M);;
```

All extractors appearing in the remaining of this section are assumed to have been defined as above.

Since Object is only a constructor, it is possible to rebuild the self object inside a method. This enables a method to return the object itself, or to send messages to itself rather than to select the right method by hand:

```
let self (R,M) = Object (R,M);;

let print (R,M)  = print_int (send getx (Object (R,M))); Object (R,M);;
```

If a points is moved, a new point must be defined with another coordinate. The only way to do this without imperative features is to return another point object with a modified state. A method move can be defined as:

```
let move (R,M) dx = Object (R || {x = R.x + dx}, M);;
```

The message getx could have been sent to the object instead of directly accessing the state, but this would not work correctly for anti-points.

Inheritance is basically sharing of methods. Most of the examples above, already illustrate some inheritance by reusing the methods pointM to build different variants of points. The following examples do this in a more systematic way.

3.3 Simple inheritance

Extending points with color points requires the extension of the state as well. For sake of simplicity, the color is represented by a boolean. The representation of points must be extended as follows:

```
let colorR superR (c) = superR || {c};;
let color_pointR (x,c) = colorR (pointR x) c;;
```

Color points should have a new method getc that returns the color. The method print had better be redefined to print the color as well. For instance, it can first print the point as before reusing the print method of points, then print the color. However, it is better to abstract on methods of points, called the super class, so that anti_points can be extended with color as well (and be printed correctly).

```
let colorM superM =
  let getc (R,M) = R.c
  and print (R,M) =
    print_string (if send getc (Object (R,M)) then "Black" else "White");
    superM.print (R,M) in
 superM || {getc; print};;
```

```
let color_pointM() = colorM pointM;;
let color_anti_pointM() = colorM anti_pointM;;
```

We have to abstract color_pointM because of the polymorphism-on-values restriction for correct typing of references [Wri93]. We exhibit here one irritating example of this restriction! If we did not allow mutable objects in the language or if we chose the original ML typing of references the abstraction would not be required.

Points and anti-points themselves might have been defined from abstract points, which, for instance, would have a move method and a default print method. Therefore, points should also have been defined by abstracting the methods of their super class. For sake of uniformity, we rewrite all definitions of method vectors by abstracting the super-class methods. Similarly, all representations should abstract over the super representation.

```
let abstract_pointM superM = ...

let pointR superR v = superR || {x = v};;
let pointM superM =
  let getx (R,M) = R.x
  in abstract_pointM superM || {getx};;
```

The representations and methods of objects can be recovered anytime by applying the representation and method generators to empty records:

```
type null = Null;;
let emptyM = ({}: {abs.'a → null}) and emptyR = ({}: {abs.null});;

let point v = Object (pointR emptyR v, pointM emptyM);;
```

Inheritance is essentially a structured method sharing. Classes are just a way of structuring inheritance. Both generator components of objects are grouped together to form a class. For convenience, we also define a null class:

```
type 'a class = Class of 'a;;

let pointC = Class (pointR, pointM);;

let nullR superR () = superR and nullM super = super;;
let nullC = Class (nullR, nullM);;
```

An object is an instance of a class with the appropriate parameters, it can be obtained shortly using the function:

```
let new (Class (classR, classM)) = let M = classM emptyM in
  fun v → Object (classR emptyR v, M);;

let point v = new pointC v;;
```

Simple inheritance can be generated in a systematic way by defining:

```
let inherits (Class (R1, M1)) (Class (R2, M2)) =
  let R superR (v1,v2) = R2 (R1 superR v1) v2 in
  let M super = M2 (M1 super) in
  Class (R, M);;
```

3.4 Multiple inheritance

Our classes are called wrappers in [Hen91b]. Each component of a class is a function that, given a record (state or methods), wraps around it its own fields (private variables or new methods). The function inherits composes the components and the function new applies the components to empty records. Lifting classes to wrappers is basically the same as lifting records to records with concatenation as done in [Rém93c]; it provides multiple inheritance for free.

Assume that a name wrapper is defined as the color one, but where color has been replaced by name of type string.

```
let colorC = Class (colorR, colorM) and nameC = ...
```

Named color points can be defined by wrapping points with either color, then name or name then color.

```
let name_color_pointC() = inherits pointC (inherits colorC nameC);;
let color_name_pointC() = inherits pointC (inherits nameC colorC);;
```

The two versions are not equivalent. For instance, the last one will print the color before the name:

```
let p1 = new (color_name_pointC()) (1, ("Board", true)) in send print p1;;
BlackBoard1− : unit = ()
```

Wrappers have replaced multiple inheritance by single inheritance. Assuming that name_pointC and color_pointC classes have been defined first, one could think of defining named color-points by inheriting from both classes. Although this is possible, it does not make much sense, since the resulting class creates two instances of points and one would overwrite the other.

3.5 Mutable objects

Objects can be programmed in the purely functional subset of ML-ART, and quite efficiently, since the record of methods can always be shared between all objects of the same class. However, movable objects must create new points each time they are moved. Clearly, there are situations when the old object becomes useless after it is moved. In this case, the object's state should be modified instead. It is quite straightforward to implement mutable objects in ML-ART using reference cells in the state of objects. For instance, mutable points could be implemented as follows:

```
let mutable_pointC =
  let pointR super x = super || {x = ref x}
  and pointM super =
    let getx (R,M) = !R.x
    and move (R,M) dx = R.x := !R.x + dx; Object (R,M)
    and print (R,M) = print_int (send getx (Object (R,M)))
    in super || {getx; move; print}
  in Class (pointR, pointM);;
```

3.6 An advanced example

In all examples we have treated so far, a class B can inherit from a class A only after B is defined. However, it may happen that some method of the super class has to create objects of an inherited class. Such a situation arises with object implementations of data types. For instance, two classes B and C have the same interface but different behaviors, while they share many methods. Thus B and C are naturally defined as inheriting from an abstract class A composed of the common methods to B and C.

However, we cannot recursively define the function that creates new objects of class B and C and the class A, since this would be an unsafe recursive definition. The simplest solution to this problem is to cut the recursion by taking some of the creation functions and putting them inside the state of objects.

For example, an abstract class listC should define all common methods to the classes nilC and consC.

```
let listC =
  let listM super =
    let map (R,M) f = let P = Object (R,M) in
      if send null P then R.new.nil ()
      else R.new.cons (f (send hd P), send map (send tl P) f)
    and print RM = let P = Object RM in
      if send null P then () else
        (send print (send hd P); send print (send tl P))
    in super || {map; print}
  in Class (nullR, listM);;
```

Then, the two classes nilC and consC inherits from class listC to which their own behaviors are added:

```
let nilC() =
  let nilM super =
    let null _ = true  in let hd _ = raise (Failure "hd") in let tl = hd
    in super || {null; hd; tl}
  in inherits listC (Class (nullR, nilM));;

let consC() =
  let consR super (h,t) = super || {h; t}
  and consM super =
    let null RM = false and hd (R,M) = R.h and tl (R,M) = R.t
    in super || {null; hd; tl}
  in inherits listC (Class (consR, consM));;
```

Last, the creators for the classes nilC and consC are recursively defined and passed to themselves. We previously define a library function new_with_new that initialize the state with a record of object creators rather than with the empty record.

```
let new_with_new (Class (classR, classM)) = let M = classM emptyM in
  fun new v → Object (classR (emptyR || {new}) v, M);;

let cons_nil() =
```

```
let new_nil = new_with_new (nilC())
and new_cons = new_with_new (consC()) in
let rec cons_nil = {nil; cons}
and nil () = new_nil cons_nil ((),())
and cons v = new_cons cons_nil ((),v) in
cons_nil;;
```

Finally we can test most programs of this section at once:

```
let {cons; nil} = cons_nil() in
let p = point 9 in
let q = send move p 75 in
let points = cons (p, cons (q, nil())) in
send print (send map points (fun x → send move x 10));;
| 1994– : unit = ()
```

3.7 Discussion

We have shown how to program most object constructions. Unfortunately, objects have no interface subtyping. That is, the ability to implicitly forget methods is lacking and objects cannot be coerced to their counterparts in super classes. The same message print can be sent to points and color points. However, both of them have incompatible types and can never be stored in the same list. Some languages with sub-typing allow this set-up. They would take the common interface of all objects that are mixed in the list as the interface of any single object of the list.

In order to be able to forget fields in ML-ART, it would be necessary to give the more general type $\{\ell : \alpha_0; \alpha_1\} \rightarrow \alpha_2 \rightarrow \{\ell : \alpha_3.\alpha_2; \alpha_1\}$ to the extension primitive $(\_ \parallel \{\ell = \_\})$. This typing is sound, but it does not provide enough polymorphism yet, because of the recursion involved in either object creation or message passing. With recursive objects (section 3.1), the record of methods is built as a fix-point and can only be assigned a monomorphic type. Non recursive objects (section 3.2) are simpler, a priori. However, forgetting any method would result in the failure to send any message to the object, since the fix-point has not been created yet.

In order to allow implicit coercions of objects to their super classes, some other kind of polymorphism must be used. Adding sub-typing could be one solution. Type-checking with non structural sub-types may find a solution along the lines of [Aik93]. Objects with type inference and subtyping but top-level class definitions have also been studied in [Chi93]. However, classes cannot be parameterized. Another interesting investigation, and probably the most promising, are type isomorphisms of Di Cosmo [DC92, DC93]. It can be expected that they would allow to turn some present flags into flag variables after the recursive objects have been created.

None of these extensions could work with non-recursive objects, which reveal too much information. That is, they disclose what messages are recursively called by other methods, which makes their types anti-monotonic in the method part.

Conversely, this information is hidden in recursive objects. After the fix-point has been taken, the method part of the type only appears in positive occurrences. This is an argument in favor of recursive objects. All examples that have been treated with non-recursive objects can very easily be adapted to recursive objects, provided some fix-points of non values can be defined.

A standard technique for compiling recursive definitions requires knowledge of the exact size of the top structure of the recursive value being defined. A dummy value of that size is allocated before evaluation of the recursive definition, whose result is used to patch the dummy value. Thus, at least the top structure of the recursive value must be statically known. Moreover, the evaluation of the recursive definition assumes that the dummy value is only passed to other functions, stored inside closures, but never accessed before it is patched. This analysis is very similar to verifying that the evaluation of some expression does not create a reference. This problem has been widely addressed recently, but has not yet found any satisfactory solution. It can be thought that any good solution for detecting creation of references can be applied to the detection of unsafe recursions as well.

In a language with references and variant types, it is possible to create recursive values by programming the schema above. However, a pre-accessing test should be performed to determine whether the value is defined. This rule should be applied even if the value was successfully defined earlier. Moreover, because of polymorphism restrictions due to the use of mutable features, this method does not apply when the recursive value must be polymorphic.

Another approach is to remark that call-by-name fix-points are always safe and not restricted to functions, and that call-by-name can be simulated with call-by-value. That is, recursive values can be replaced by recursive abstractions on values, which can be safely defined. This solution has been proposed in [Pie93]. However, extra abstractions stop evaluation. Consequently, method vectors are rebuilt every time a message resends another message to itself. This is too inefficient. Moreover, call-by-value runtime errors (unsafe examples) have been changed into call-by-name "safe" loops. Is this more satisfactory?

Other solutions could require annotations of the source code to help the static analyzer. There are easy solutions that would automatically guarantee safety of the above examples. All of them are still more or less *ad hoc*, therefore none of them has been included into the language ML-ART.

If extending the type system to provide some inclusion is a prerequisite to programming objects with interface subtyping, a clean and efficient solution should also extend fix-points to allow some restricted form of recursion on non-values.

Conclusion

Programming objects with ML-ART is an interesting experiment, that primarily helps to understand objects in several ways. The fundamental feature in object-oriented programming is message passing. Polymorphic access is required and it suffices to model very simple objects. Subsequently, it is necessary to conceal the internal state of objects, either by value or type abstraction. Although more difficult, type abstraction is much more efficient and has proved to be feasible. The concept of inheritance is essentially structured method sharing. Polymorphic record extension is sufficient for simple and multiple inheritance. Finally, we determine that, classes are just a way of structuring inheritance.

As opposed to the encoding in $F^{\omega}_{<:}$ that requires a lot of systematic, but still necessary, type information, all examples could be written in a natural ML style. This allows us to assert that no syntactic construct is needed for programming objects in ML-ART. Programmable objects are easier to understand than primitive objects: there is no need to learn a new language. Instead, object-oriented programming can be discovered progressively.

We have presented one programming style for objects but other interesting ones can certainly be found. Some of them could be offered in libraries to allow the user the choice of object complexity, that is consistent with the level of his problem. A beginner would probably adopt a style from the library while an expert would define his own.

The language ML-ART is a powerful extension to ML. Record types make declarations of record data structures optional. Although recursive types may be quite useful in a few other circumstances, existential and universal types, through type declarations, seem to possess the degree of higher-orderness needed in practice. Type information, carried by constructors, keeps the language very close to ML and makes it as easy to use.

The main limitation of our objects is their inability to be coerced to corresponding objects of super classes. Improvements of the type system should be made to address this problem, upon finding a satisfactory solution to the second problem of non-functional recursive values. Both of these issues are interesting and worth further investigation.

Acknowledgments

I am indebted to Benjamin Pierce for convincing me that abstract types were the correct approach to objects, and for generating many fruitful discussions. This work evolved during a two-month visit to Bell Labs, through seminar discussions on objects. An earliest version of objects was also written during this visit. I am also thankful to Martin Abadi, Luca Cardelli and Lucky Chillan whose comments were very helpful.

References

[AC91] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages*, pages 104–118, Orlando, FL, January 1991. Also available as DEC Systems Research Center Research Report number 62, August 1990.

[AC94] Martin Abadi and Luca Cardelli. A theory of primitive objects. In *International Symposium on Theoretical Aspects of Computer Software*, April 1994.

[Aik93] Alexander Aiken. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture*, pages 31–41. ACM press, 1993.

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

[Bru93] Kim B. Bruce. Safe type checking in a statically typed object-oriented programming language. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–68. Springer Verlag, 1984. Also in Information and Computation, 1988.

[Chi93] Lucky Chillan. *Une extension de ML avec des aspects orientés objets.* Thèse de doctorat, Université de Paris 7, Place Jussieu, Paris, France, 1993. Forthcoming.

[CM89] Luca Cardelli and John C. Mitchell. Operations on records. In *Fifth International Conference on Mathematical Foundations of Programming Semantics*, 1989.

[DC92] Roberto Di Cosmo. Deciding type isomorphisms in a type assignment framework. *Journal of Functional Programming*, 1992. To appear in the Special Issue on ML.

[DC93] Roberto Di Cosmo. *Isomorphisms of Types.* Tesi di dottorato, Dipartimento di Informatica, Universitá di Pisa, 40, Corso Italia - 56100 Pisa - Italy, January 1993.

[GRR93] Carl Gunter, Didier Rémy, and John Riecke. Syntactic type soundness with prompt, callcc and state. Manuscript, 1993.

[Hen91a] Andreas V. Hense. An O'small interpreter based on denotational semantics. Technical Report A 07/91, Universitıat des Saarlandes, Fachbereich 14, 1991.

[Hen91b] Andreas V. Hense. Wrapper semantics of an object oriented programming language with state. *Theoretical Aspects of Computer Science*, Lecture notes in Computer Science(526), September 1991.

[HMT91] Robert Harper, Robin Milner, and Mads Tofte. *The definition of Standard ML.* The MIT Press, 1991.

[HP90] Robert W. Harper and Benjamin C. Pierce. Extensible records without subsumption. Technical Report CMU-CS-90-102, Carnegie Mellon University, Pittsburg, Pensylvania, February 1990.

[HP92] Martin Hofmann and Benjamin Pierce. An abstract view of objects and subtyping (preliminary report). Technical Report ECS-LFCS-92-226, University of Edinburgh, LFCS, 1992.

[Hue76] Gérard Huet. *Résolution d'équations dans les langages d'ordre* $1, 2, \ldots, \omega$. Thèse de doctorat d'état, Université Paris 7, 1976.

[JM88] Lalita A. Jategaonkar and John C. Mitchell. ML with extended pattern matching and subtypes (preliminary version). In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 198–211, Snowbird, Utah, July 1988.

[Läu92] Konstantin Läufer. *Polymorphic Type Inference and Abstract Data Types.* PhD thesis, New York University, 1992.

[LM92] Xavier Leroy and Michel Mauny. The caml light system, version 0.5. documentation and users' guide. Logiciel 3, INRIA-Rocquencourt, BP 105, F-78 153 Le Chesnay Cedex, 1992.

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

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

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[OB88] Atsushi Ohori and Peter Buneman. Type inference in a database langage. In *ACM Conference on LISP and Functional Programming*, pages 174–183, 1988.

[Oho90] Atsushi Ohori. Extending ML polymorphism to record structure. Technical report, University of Glasgow, 1990.

[Pie93] Benjamin C. Pierce. Mutable objects. Unpublished note, June 1993.

[PT93] Benjamin C. Pierce and David N. Turner. Object-oriented programming without recursive types. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.

[Rém92a] Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique, BP 105, F-78 153 Le Chesnay Cedex, 1992.

[Rém92b] Didier Rémy. Projective ML. In *1992 ACM Conference on Lisp and Functional Programming*, pages 66–75, New-York, 1992. ACM press.

[Rém93a] Didier Rémy. Syntactic theories and the algebra of record terms. Research Report 1869, Institut National de Recherche en Informatique, BP 105, F-78 153 Le Chesnay Cedex, 1993.

[Rém93b] 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, 1993. To appear.

[Rém93c] Didier Rémy. Typing record concatenation for free. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993. To appear.

[Wan87] 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.

[WF91] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Rice University, 1991.

[Wri93] Andrew K. Wright. Polymorphism for imperative languages without imperative types. Technical Report 93-200, Rice University, February 1993.

Definition of the language ML-ART

We describe an extended language with locations and record values so that we can give a reduction semantics inside the language itself [WF91].

A Expressions

The syntax of expressions is given below. We assume given a denumerable set of variables and a denumerable set of locations. Letter x ranges over variables and letter l ranges over locations.

$a ::= x \mid c \mid \mathsf{fun}\ x \rightarrow a \mid a_1\ a_2$	Expressions
$\quad \mid \mathsf{let}\ K\ x = a_1\ \mathsf{in}\ a_2 \mid \mathsf{strip}\ K\ \mathsf{of}\ a \mid K\ a$	
$\quad \mid \{\vec{\ell} = \vec{v}\} \mid \mathrm{loc}\, l$	
$c ::= c_s \mid c_d$	Constants
$c_s ::= (\_ \parallel \{\ell = \_\}) \mid (\_.\ell) \mid (!\_) \mid (\_ := \_) \mid \ldots$	Safe constants
$c_d ::= \mathrm{ref}$	Dangerous constants
$v ::= c \mid \mathsf{fun}\ x \rightarrow a \mid \{\vec{\ell} = \vec{v}\} \mid K\ v \mid \mathrm{loc}\, l$	Values
$b ::= x \mid c_s\ \vec{b} \mid c_d \mid \mathsf{fun}\ x \rightarrow a$	Generalizable terms
$\quad \mid \mathsf{let}\ K\ x = b\ \mathsf{in}\ b \mid \mathsf{strip}\ K\ \mathsf{of}\ b$	
$\quad \mid K\ b \mid \{\vec{\ell} = \vec{b}\}$	

We consider expressions modulo renaming of bound variables. Record values are finite maps from labels to values, and are taken modulo reordering of fields. Record values, as well as locations, are added to the language to permit a reduction semantics. Otherwise, only the empty record $\{\}$ would be given as a constant. The extension primitive $(\_ \parallel \{\ell = \_\})$ can be used to define records that are not values. In the implementation of ML-ART, we allow $\{\vec{\ell} = a\}$ in the syntax but expand it immediately into a sequence of extensions when a is not a value.

For convenience, we allow duplication of fields in the notation, with priority to the right. For instance, in $\{\ell_1 = a_1, \ldots \ell_n = a_n, \ell = a\}$, label ℓ may be one of the ℓ_i, but field ℓ is always mapped to a.

Generalizable terms extend values while preserving the property that their evaluation will never create any location. They are used in the definition of typing rules.

The ML polymorphic binding $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$ can be seen as syntactic sugar for $\mathsf{let}\ K_0\ x = K_0\ a_1\ \mathsf{in}\ a_2$ for some degenerated existential constructor K_0. The expression $\mathsf{strip}\ K\ \mathsf{of}\ a$ is a simpler form of $\mathsf{let}\ Kx = Ka\ \mathsf{in}\ x$ for universal bindings. For sake of simplicity, type definitions can be considered as predefined in the initial environment (see appendix 2).

B Sorts and types

Types are defined relatively to a set of K type symbols and a set of Ω type symbols. Their syntax is given below.

$\tau ::= \alpha \mid \tau \to \tau \mid \kappa(\tau) \mid \Omega(\tau) \mid \mathsf{rec}\,\alpha.\tau \mid \{\tau\}$	Types
$\quad \mid \tau.\tau \mid (\ell : \tau; \tau) \mid [\tau] \mid \mathsf{abs} \mid \mathsf{pre}$	
$\varsigma ::= \mathsf{Usual} \mid \mathsf{Field} \mid \mathsf{Flag}$	Kind sorts
$\pi ::= \mathsf{Type} \mid \mathsf{Row}\,(L)$	Power sorts
$L ::= \emptyset \mid \ell.L \qquad \ell \notin L$	

The formation of types is restricted twice by kind sorts and power sorts. Kind (respectively power) signatures are non empty sequences of kind (respectively power) sorts, written $\vec{\varsigma_i} \Rightarrow \varsigma$ or just ς when $\vec{\varsigma_i}$ is empty. Each type symbol comes with both a kind signature and a power signature:

Symbols	Kinds	Powers
κ	$\mathsf{Usual} \Rightarrow \mathsf{Usual}$	$\pi \Rightarrow \pi$
$(\_ \to \_)$	$(\mathsf{Usual}, \mathsf{Usual}) \Rightarrow \mathsf{Usual}$	$(\pi, \pi) \Rightarrow \pi$
$\{\_\}$	$\mathsf{Field} \Rightarrow \mathsf{Usual}$	$\mathsf{Row}\,(\emptyset) \Rightarrow \mathsf{Type}$
$\_.\_$	$(\mathsf{Flag}, \mathsf{Usual}) \Rightarrow \mathsf{Field}$	$(\pi, \pi) \Rightarrow \pi$
$(\ell : \_; \_)$	$(\varsigma, \varsigma) \Rightarrow \varsigma$	$(\mathsf{Type}, \mathsf{Row}\,(\ell.L)) \Rightarrow \mathsf{Row}\,(L)$
abs, pre	Flag	π
$[\_]$	$\varsigma \Rightarrow \varsigma$	$\mathsf{Type} \Rightarrow \mathsf{Row}\,(L)$

Sort metavariables in signatures mean that all forms ranged over by this metavariable are possible. Thus a symbol may have several signatures. However, for any term and any sort, there is at most one possible assignment of signatures to symbol occurrences such that the term is well sorted. There is an algorithm that tests whether such an assignment exists and if it does, that computes it. Thus, it would be possible to decorate types so that they form a many-sorted algebra in the usual meaning.

The Ω type symbols are introduced without their signature, since they can always be inferred unambiguously from the sorts of the existential variables that they substitute.

The most significant sorts are the kinds. They avoid using flags in positions of usual types. All types appearing in typing rules and in type environments have the kind Usual and the power Type. Expressions of power $\mathsf{Row}\,(L)$ are templates in record types and L enumerates all labels that the template must not define. This is used to avoid redefinition of fields in record types.

In fact, the above sorts allow such type expressions as $\{\ell_1 : \mathsf{abs}.\alpha_1; (\ell_2 : \alpha_2.\mathsf{abs}).(\alpha_3 \to \alpha_4)\}$ but types that the user can see utilize only the weaker kind signature $(\mathsf{Flag}, \mathsf{Usual}) \Rightarrow \mathsf{Field}$ for the $(\ell : \_ ; \_)$ symbol, which forbids such types as above.

We write $\{\alpha \leftarrow \tau\}$ the substitution that replaces free occurrences of α by τ. We often write $\vec{\alpha}$ for tuples of variables.

Type equality is the smallest congruence that satisfies the equations of the projective algebra and those for recursive types. Type equations of the projective algebra are, for any type symbol f other than $(\ell : \_ ; \_)$, $[\_]$, and $\{\_\}$, for any labels ℓ_1, ℓ_2, and ℓ,

$$(\ell_1 : \alpha_1 ; \ell_2 : \alpha_2 ; \alpha_0) = (\ell_2 : \alpha_2 ; \ell_1 : \alpha_1 ; \alpha_0)$$

$$f\overrightarrow{(\ell : \alpha_1 ; \alpha_2)} = (\ell : f(\vec{\alpha}_1) ; f(\vec{\alpha}_2)) \qquad [\alpha] = (\ell : \alpha ; [\alpha]) \qquad [f(\vec{\alpha})] = f\overrightarrow{[\alpha]}$$

The equational theory of projective types (without recursive types) is regular and collapse-free, but not linear. It is studied and proved syntactic in [Rém93a].

The recursive type expression $(\mathsf{rec}\,\alpha.\tau)$ is well formed only if both α and τ have the kind sort Usual and power sort Type and if τ is neither a type variable, nor another $(\mathsf{rec}\,\_.\_)$. This guarantees τ is contractive in α and that $\mathsf{rec}\,\alpha.\tau$ is well-defined [AC91]. Equality for recursive types is taken from [AC91] (all types are assumed to be well-sorted):

$$\tau_0 = \tau_1 \Longrightarrow \mathsf{rec}\,\alpha.\tau_0 = \mathsf{rec}\,\alpha.\tau_1 \qquad (\textsc{Congruence})$$

$$\mathsf{rec}\,\alpha.\tau = \tau\{\alpha \leftarrow \mathsf{rec}\,\alpha.\tau\} \qquad (\textsc{Fold-Unfold})$$

$$\tau_1 = \tau\{\alpha \leftarrow \tau_1\} \wedge \tau_2 = \tau\{\alpha \leftarrow \tau_2\} \Longrightarrow \tau_1 = \tau_2 \qquad (\textsc{Contract})$$

Of course, $\mathsf{rec}\,\_.\_$ acts as a binder in the first argument and variable α is not free in $\mathsf{rec}\,\alpha.\tau$.

The following property asserts that type equality does not identify too many types. It is used in the proof of theorem 5 below.

Proposition 1 type-consistency. *Any type of kind* Usual *and power* Type *that is equal to a type* $f(\vec{\tau}_1)$ *is syntactically equal to either* $f(\vec{\tau}_2)$ *or* $\mathsf{rec}\,\alpha.f(\vec{\tau}_2)$ *for some terms* $\vec{\tau}_2$.

C Typing rules

Type schemes and type assignment formulas are given in figure 1. The following type declarations are not expressions of the language:

$$\mathsf{type}\ \kappa(\tau_0) = K\ \mathsf{of\ Exist}\ (\vec{\alpha})\ \tau \qquad \text{or} \qquad \mathsf{type}\ \kappa(\tau_0) = K\ \mathsf{of\ All}\ (\vec{\alpha})\ \tau$$

They are replaced by type assignment:

$$K : \forall\, \vec{\alpha}_0.\, \mathsf{Exist}(\vec{\alpha})\, \tau \mathrel{-\!\!\rhd} \kappa(\tau_0) \qquad \text{or} \qquad K : \forall\, \vec{\alpha}_0.\, \mathsf{All}(\vec{\alpha})\, \tau \mathrel{-\!\!\rhd} \kappa(\tau_0) \qquad (1)$$

We say that type constructor K and type symbol κ are paired in type assignment (1). The expression $\mathsf{Exist}(\vec{\alpha})\, \tau \mathrel{-\!\!\rhd} \tau_0$ and $\mathsf{All}(\vec{\alpha})\, \tau \mathrel{-\!\!\rhd} \tau_0$ are well-formed if

$$
\begin{array}{ll}
\sigma ::= \tau \mid \mathsf{Exist}(\vec{\alpha})\,\tau_1 \rightarrowtriangle \tau_2 \mid \mathsf{All}(\vec{\alpha})\,\tau_1 \rightarrowtriangle \tau_2 & \text{Type schemes} \\
\quad \mid \forall\,\alpha.\,\sigma & \\
 & \\
A ::= \emptyset \mid A[l : \tau] \mid A[x : \sigma] \mid A[c : \sigma] & \text{Type environments} \\
\quad \mid A[K : \sigma] \mid A[\Omega] & \\
\Delta ::= A \vdash a : \sigma \mid A \vdash K : \sigma \mid A \vdash \sigma & \text{Judgements}
\end{array}
$$

Fig. 1. Type assignment formulas

- $\vec{\alpha}$ is linear, i.e. no variable occurs twice,
- variables $\vec{\alpha}$ are not in τ_0,
- all variables of τ occur in either $\vec{\alpha}$ or τ_0.

Type scheme $\forall\,\alpha.\,\sigma$ is well-formed if σ is. We abbreviate sequences of quantifiers $\forall\,\alpha_1.\cdots\forall\,\alpha_n.\,\sigma$ by $\forall\,\alpha_1,\cdots\alpha_n.\,\tau$.

Well-formed type environments are recursively defined as follows. The empty environment is well-formed. The environment $A[\Omega]$ is well-formed if A is well-formed and does not introduce Ω. The environments $A[\_ : \sigma]$ are well-formed if A is and if all symbols of σ are predefined or introduced in A. Last, type assignment formula $A \vdash \_ : \sigma$ is well-formed if the environment $A[\_ : \sigma]$ is.

We assume that an initial environment A_0 assigns types schemes to constants as given below and type schemes of the form (1) to type constructors such that each type constructor K is paired with at most one type symbol κ in A_0.

$$
\begin{array}{r@{\;}l}
(\_.\ell) : & \forall\,\alpha_1,\alpha_2.\,\{\ell : \mathsf{pre}.\alpha_1; \alpha_2\} \rightarrow \alpha_1 \\
(\_ \mid\mid \{\ell = \_\}) : & \forall\,\alpha_1,\alpha_2,\alpha_3.\,\{\ell : \alpha_1; \alpha_2\} \rightarrow \alpha_3 \rightarrow \{\ell : \mathsf{pre}.\alpha_3; \alpha_2\} \\
(\mathrm{ref}\,\_) : & \forall\,\alpha.\,\alpha \rightarrow \mathrm{ref}\,(\alpha) \\
(!\_) : & \forall\,\alpha.\,\mathrm{ref}\,(\alpha) \rightarrow \alpha \\
(\_ := \_) : & \forall\,\alpha.\,\mathrm{ref}\,(\alpha) \rightarrow \alpha \rightarrow \alpha
\end{array}
$$

We write $\mathcal{V}(A)$ for all variables of A.

Typing rules are given in figure 2. Variable z ranges over identifiers x, c, and K. Rule Exist and All should be seen as existential and universal introduction rules. The Strip rule is clearly the opposite of All and corresponds to universal elimination. We can see the expression $\mathsf{let}\ K\ x = a_1\ \mathsf{in}\ a_2$ that we used in section 2.5 as syntactic sugar for $\mathsf{let}\ x = \mathsf{strip}\ K\ \mathsf{of}\ a_1\ \mathsf{in}\ a_2$. The same simplification cannot be used for existential elimination because the above transformation would break the scope of the Ω's introduced by $\mathsf{strip}\ K\ \mathsf{of}\ a_1$. Rule Exist is a combination of existential elimination and generic binding. The expression $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$ is not in the language, but it can be added as syntactic sugar for $\mathsf{let}\ K_0\ x = K_0\ a_1\ \mathsf{in}\ a_2$ where type assignment $K_0 : \forall\,\alpha.\,\mathsf{Exist}()\,\alpha \rightarrowtriangle \alpha$ is assumed to be in A_0.

$$\frac{z : \forall \vec{\alpha}_j .\, \tau \in A}{A \vdash z : \tau\{\vec{\alpha}_j \leftarrow \vec{\tau}_j\}}\ (\textsc{Get})$$

$$\frac{A \vdash b : \sigma \qquad \alpha \notin \mathcal{V}(A)}{A \vdash b : \forall \alpha .\, \sigma}\ (\textsc{Gen}) \qquad \frac{l : \tau \in A}{A \vdash \mathrm{loc}\, l : \mathrm{ref}\,(\tau)}\ (\textsc{Loc})$$

$$\frac{A[x : \tau_0] \vdash a : \tau_1}{A \vdash \mathsf{fun}\ x \rightarrow a : \tau_0 \rightarrow \tau_1}\ (\textsc{Fun}) \quad \frac{A \vdash a_1 : \tau_1 \rightarrow \tau_0 \qquad A \vdash a_2 : \tau_1}{A \vdash a_1\ a_2 : \tau_0}\ (\textsc{App})$$

$$\frac{A \vdash v_1 : \tau_1 \qquad \ldots \qquad A \vdash v_n : \tau_n}{A \vdash \{\ell_1 = v_1, \ldots \ell_n = v_n\} : \{\ell_1 : \mathsf{pre}.\tau_1; \ldots \ell_n : \tau_n; \mathsf{abs}.\alpha\}}\ (\textsc{Record})$$

$$\frac{A \vdash a : \tau_0 \qquad A \vdash K : \mathsf{Exist}(\vec{\tau})\, \tau_0 \rightarrowtriangle \tau_1}{A \vdash K\ a : \tau_1}\ (\textsc{Exist})$$

$$\frac{A \vdash a : \forall \vec{\alpha} .\, \tau_0 \qquad A \vdash K : \mathsf{All}(\vec{\alpha})\, \tau_0 \rightarrowtriangle \tau_1}{A \vdash K\ a : \tau_1}\ (\textsc{All})$$

$$\frac{A \vdash a_1 : \tau_1 \qquad A \vdash K : \mathsf{All}(\vec{\tau})\, \tau_0 \rightarrowtriangle \tau_1}{A \vdash \mathsf{strip}\ K\ \mathsf{of}\ a_1 : \tau_0}\ (\textsc{Strip})$$

$$\frac{\begin{array}{c} A \vdash K : \forall \vec{\alpha}_1, \vec{\alpha}_j .\, \mathsf{Exist}(\vec{\alpha}_j)\, \tau_0 \rightarrowtriangle \tau_1 \\ A \vdash a_1 : \forall \vec{\alpha}_1 .\, \tau_1 \qquad A[\vec{\Omega}_j][x : \forall \vec{\alpha}_1 .\, \tau_0\{\vec{\alpha}_j \leftarrow \vec{\Omega}_j(\tau_1)\}] \vdash a_2 : \tau_2 \end{array}}{A \vdash \mathsf{let}\ K\ x = a_1\ \mathsf{in}\ a_2 : \tau_2}\ (\textsc{Let})$$

Fig. 2. Typing rules.

In a derivation of a typing judgement, GEN rules can only be used as the last ones or before the left hand sides of LET and FORALL rules, since these are the only premises that allow type schemes. We write GEN* for a possibly empty sequence of GEN rules. Moreover, we can always assume that it is used as much as possible on the left hand sides of LET rules. We call such derivations canonical.

In the next section we will use the following properties of typings.

Proposition 2 Stability by substitution. *If $A \vdash a : \tau$ then $\mu(A) \vdash a : \mu(\tau)$ for any substitution μ such that the formula is well-formed.*

Proposition 3 Extension of environment. *If the type-assignement A and B are identical everywhere except maybe on variables that are not free in a, then $A \vdash a : \sigma$ is derivable if and only if $B \vdash a : \sigma$ is.*

These properties are proved in [Rém92a] for core ML when types are taken modulo a regular equational theory. A regular theory is one such that two equal terms always have the same free variables. All equations for the projective algebra and for recursive types are regular. The proofs of [Rém92a] easily extend to the language ML-ART.

In [Rém92a] we also show that the language has principal typing if the equational theory has principal unifiers. The proofs extend to all constructions of ML-ART. It is proved in [Rém93a] that the equational theory of recursive types has principal unifiers. We have not verified that the combination of the theories of projective types and recursive types have principal types, but we conjecture so. Thus, the algorithm for type inference without recursive types is sound with recursive types but it may not be complete.

D Semantics

We give a call-by-value reduction semantics of ML-ART using the general formalism [WF91] and treating the store as in [GRR93]. We define stores as finite mappings from locations to values. Call-by-value evaluation contexts are:

$$\begin{aligned} E ::= & \{\} \mid \mathsf{let}\ K\ x = E\ \mathsf{in}\ a \mid K\ E \mid \mathsf{strip}\ K\ \mathsf{of}\ E \mid E\ a \mid v\ E \\ & \mid \mathrm{ref}\ E \mid !E \mid E := a \mid v := E \mid E \parallel \{\ell = a\} \mid v \parallel \{\ell = E\} \mid E.\ell \end{aligned}$$

The semantics is given by a step reduction relation $\longrightarrow$:

	$(\mathsf{fun}\ x \rightarrow a)\ v/s \longrightarrow a\{x \leftarrow v\}/s$	FUN
	$\mathsf{let}\ K\ x = K\ v\ \mathsf{in}\ a/s \longrightarrow a\{x \leftarrow v\}/s$	LET
	$\mathsf{strip}\ K\ \mathsf{of}\ K v/s \longrightarrow v/s$	STRIP
	$\{\vec{\ell_i} = \vec{v_i}\} \parallel \{\ell = v\}/s \longrightarrow \{\vec{\ell_i} = \vec{v_i}; \ell = v\}/s$	WITH
	$\{\vec{\ell_i} = \vec{v_i}; \ell = v\}.\ell/s \longrightarrow v/s$	DOT
If $l \notin dom(s)$,	$\mathrm{ref}\ v/s \longrightarrow \mathrm{loc}\ l/s[l \mapsto v]$	REF
If $l \in dom(s)$,	$!(\mathrm{loc}\ l)/s \longrightarrow s(l)/s$	DEREF
If $l \in dom(s)$,	$\mathrm{loc}\ l := v/s \longrightarrow v/s[l \mapsto v]$	ASSIGN
If $a_1/s_1 \longrightarrow_\epsilon a_2/s_2$,	$E\{a_1\}/s_1 \longrightarrow E\{a_2\}/s_2$	CONTEXT

We say that store s agrees with type environment A, and we write $\Vdash s : A$ if both s and A have the same location domains, and for any location l of their domain $A \vdash s(l) : A(l)$. We call a store extension of A an extension of A with any number of location type assignements $l : \tau$. We write $a_1/s_1 \subset a_2/s_2$ if

- for any environment A_1, any type τ such that $A_1 \vdash a_1 : \tau$ and $\Vdash s_1 : A_1$, there exists a store extension A_2 of A_1 such that $A_2 \vdash a_2 : \tau$ and $\Vdash s_2 : A_2$,
- a_2 is generalizable whenever a_1 is and then A_2 may be chosen equal to A_1.

The soundness of the semantics is formalized by the two following theorems:

Theorem 4 Subject Reduction. *If $a_0/s_0 \longrightarrow a/s$ then $a_0/s_0 \subset a/s$.*

Theorem 5 Normal forms. *Let A be a store extension of the initial environment A_0. If $A \vdash a : \tau$ and $\vdash s : A$ and a/s is in $\longrightarrow$-normal form, then a is a value.*

Subject reduction is a straightforward combination of redex contraction and context replacement lemmas.

Lemma 6 Context replacement. *For any one-hole context E, if $a_1/s_1 \subset a_2/s_2$ then $E\{a_1\}/s_1 \subset E\{a_2\}/s_2$.*

By construction, the relation $\subset$ is reflexive, transitive; context replacement says that it is also increasing. The lemma is proved independently for each one-nod context, then the general case follows by induction on the size of the context.

Lemma 7 Redex contraction. *If $a_1/s_1 \longrightarrow_\epsilon a_2/s_2$ then $a_1/s_1 \subset a_2/s_2$.*

The proof can be done independently for each redex. All cases are easy once we have proven the right lemmas.

Lemma 8 Term replacement. *If the formulas $A \vdash b : \forall \alpha_0 . \tau_0$ and $A[x : \forall \alpha_0 . \tau_0] \vdash a : \tau$ are provable and if bound variables of a are not free in b, then $A \vdash a\{x \leftarrow b\} : \tau$ is provable.*

Lemma 9 Existential elimination. *If $A[\Omega_j][x : \forall \alpha_0 . \tau_1\{\vec{\alpha}_j \leftarrow \vec{\Omega}_j(\tau_0))\}] \vdash a : \tau$, and $\vec{\tau}_j$ are terms whose variables are also variables of τ_0 then the formula $A[x : \forall \vec{\alpha}_0 . \tau_1\{\alpha_j \leftarrow \tau_j\}] \vdash a : \tau$ is valid whenever it is well-formed.*

The second theorem asserts that well-typed terms that cannot be reduced are values, thus the evaluation is never "stuck." It is proved by structural induction on the value using the following lemma (which itself uses the type-consistency property 1).

Lemma 10. *Let A be a store extension of A_0 such that $A \vdash v : \tau$.*

- *if τ is a functional type then a is a function or a constant.*
- *if τ is a record type then v is a record; moreover, if τ is of the shape $\{\ell : \mathsf{pre}.\tau_1; \tau_2\}$, field ℓ is defined.*
- *if τ is $\kappa(\tau_1)$ then v is a value $K v_1$ where K and κ are paired in A.*
- *if τ is $\mathrm{ref}\,(\tau_1)$ then v is a location.*

This article was processed using the LaTeX macro package with LLNCS style