Simple objects for Standard ML

John Reppy
AT&T Research
jhr@research.att.com

Jon Riecke
Bell Laboratories
riecke@research.att.com

ABSTRACT

We propose a new approach to adding objects to Standard ML (SML) based on explicit declarations of object types, object constructors, and subtyping relationships, with a generalization of the SML case statement to a "typecase" on object types. The language, called Object ML (OML), has a type system that conservatively extends the SML type system, preserves sound static typing, and permits type inference. The type system sacrifices some of the expressiveness found in recently proposed schemes, but has the virtue of simplicity. We give examples of how features found in other object-oriented languages can be emulated in OML, discuss the formal properties of OML, and describe some implementation issues.

Keywords: Programming language design, object-oriented programming, functional programming, Standard ML.

1 Introduction

A fundamental tension of language design is "keeping the language small" versus "providing support for common idioms." This fundamental tension can be seen clearly in Standard ML (SML) [MTH90]. Even though SML provides many powerful features, such as higher-order functions, polymorphism and parameterized modules, it does not provide primitives for "*object-oriented*" programming. In particular, there is no support for subtyping or for working with heterogeneous collections. Programmers must use explicit coercions to implement subtyping (for example, see [GR93]), and use tagged unions to implement heterogeneous collections.

This paper describes a simple extension to SML, called *Object ML* (OML), that provides direct support for objects, subtyping, and heterogeneous collections. Our design sacrifices expressiveness for simplicity. To aid type inference (so that programmers need not specify the types of variables), OML forces the programmer to declare object types, object constructors, and subtyping relationships between object types. OML is a conservative extension of SML, i.e., existing SML programs have the same types and semantics in OML. The object constructs in OML also have efficient constant-time implementations; there are no hidden costs, such as dynamic searching for methods.

Outside of the theoretical community, most statically typed object-oriented languages have only first-order functions, e.g., C++ [Str94] or Modula-3 [Nel91]. Even with this limitation on functions, the type systems often are too inflexible to support the object-oriented idioms found in dynamically typed languages like CLOS [Ste90], Dylan [App92], Obliq [Car95], Self [US87], or Smalltalk [GR83]). This inflexibility is overcome by using some form of dynamically checked type casting. The design of more flexible and safe type systems for languages with objects, subtyping, and higher-order functions is especially challenging, since function subtyping introduces subtle technical problems. Previous attempts require substantial type-theoretic machinery and complex type systems (see [FM94] for a review of the issues); furthermore, in some cases type checking is undecidable, and for most others, type inference seems out of the question.

The complexity of these systems make them unsatisfactory for extending SML with object-oriented features. We avoid the complexities of these systems and preserve SML's type inference mechanism, while still providing an expressive notation for object-oriented programming. Our approach is based on adding object type declarations, which provide both an explicitly declared subtyping hierarchy and named object constructors. Just as datatype constructors guide type inference in the presence of recursive types, object type constructors guide type inference in the presence of subtyping. While the system is not as expressive as those discussed above, it is more expressive than the type systems of C++ and Modula-3. In addition to supporting polymorphism, our system supports methods that return self,

whereas C++ and Modula-3 fix the return types of their methods. Furthermore, the system generalizes the SML case statement to be a typecase on object types, which can be used to recover type information dynamically that the type system cannot determine statically.[1]

We first describe our approach in a simple language with "single inheritance" of object types, where objects are records. We then complete the description of OML by introducing methods and self types. Section 4 discusses the interactions between our extensions and the features of SML. In Section 5, we describe how various object-oriented idioms are programmed in OML, followed by a discussion of implementation inheritance. Section 7 gives an overview of the theoretical foundations of OML. In Section 8, we discuss compile-time and run-time implementation issues. Finally, we discuss related work and present conclusions.

2 Extending SML with simple objects

SML's approach to recursive types guides the design of OML. The SML type system is based on the Damas-Milner type inference system [DM82], which defines a *structural* view of type equivalence (i.e., two types are equal if they have the same structure). One can extend this structural view to recursive types. For example, the type of integer lists might be defined as

$$intlist \quad \equiv \quad \mathbf{rec}\, t\,.\,(\mathbf{nil} + \mathbf{cons} : (int \times t))$$

Two recursive types are equivalent if they unfold to the same infinite tree. Unfortunately, we encounter problems if we extend this to recursive type constructors (e.g., "α `list`"), because there is no known algorithm for determining type equality in such a setting [Sol78], and type equality is at the core of the Damas-Milner type inference system. To avoid this problem, SML introduces *datatype* definitions, where each definition defines a new unique type. For example, a list type constructor can be defined as:

```
datatype 'a list = Nil | Cons of ('a * 'a list)
```

This definition introduces two *constructors*, `Nil` and `Cons`, that serve as syntactic signposts for the type inference algorithm. In patterns, constructors unfold the recursive type, while in expressions they fold the recursive type. With this information, the type inference problem for recursive types is trivial.

We adopt the same trick to deal with type inference for object subtyping. Our mechanism is based on a new declarative construct:

objtype *tyid* **=** *conid* **of {|**
lab_1 **:** ty_1**,** ..., lab_n **:** ty_n
|}

This declaration defines a new object type *tyid* with corresponding object constructor *conid*. The $lab_1 \ldots, lab_n$ are the *members* of the object type, which must be distinct. It is also possible to define an object type in terms of an existing object type:

objtype *tyid'* **is** *tyid* **=** *conid'* **of {|**
lab_{n+1} **:** ty_{n+1}**,** ..., lab_m **:** ty_m
|}

This declaration defines a new object type *tyid'* that extends the object type *tyid* with the additional members $lab_{n+1} \ldots, lab_m$. In addition to defining a new object type, this form specifies that *tyid'* is a subtype of *tyid*; the types of the members $lab_1, \ldots, lab_n$ are the same in both object types. We use C++ terminology, and call *tyid* a *base* object type, and *tyid'* a *derived* object type. These declarative forms provide a mechanism for "single inheritance" of interface types.[2] For example, the following declarations define a type hierarchy for points and colored points in $\Re^1$ and $\Re^2$:

```
objtype point1 = PT1 {| x : real |}
objtype point2 is point1 = PT2 {| y : real |}
objtype cpoint1 is point1 = CPT1 {| c : color |}
objtype cpoint2 is point2 = CPT2 {| c : color |}
```

Note that unlike in most record subtyping systems, the types `cpoint1` and `cpoint2` are not related in this hierarchy, as can be seen in Figure 1.

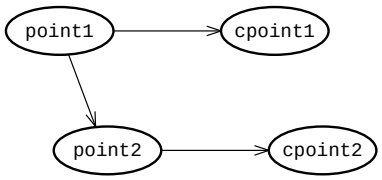

Figure 1: The type hierarchy of points and color points

Subtyping in OML is defined inductively. We start with the explicit hierarchy defined by the **`objtype`** definitions, and extend it to the various ML type constructors (tuples, labeled records and functions types) in the standard way. As is usual, subtyping on function types is contravariant in the negative position (argument position):

$$\frac{\sigma \prec \tau \quad \tau' \prec \sigma'}{\tau \to \tau' \prec \sigma \to \sigma'}$$

It is this property of subtyping that causes many of the technical difficulties in type systems for object-oriented languages.

Objects are created by applying an object constructor to a collection of member definitions. For example, the following expressions create a `point1` and a `cpoint2` object:

```
PT1{|x = 1.0|}
CPT2{|x = 1.0, y = 1.0, c = red|}
```

[1] This is our form of dynamically checked casting.

[2] We use the term "single inheritance" loosely here, since we are talking about interfaces and not implementations.

The member definitions may be specified in any order, but are evaluated left-to-right.

Just as pattern matching is used to deconstruct recursive types in SML, we use it to access the members of an object, e.g.,

```
fun xCoord (PT1{|x|}) = x
fun color (CPT1{|c, ...|}) = c
```

(the "`...`" notation means ignore the other members). An object constructor pattern can match any subtype of the object type: for example, the `xCoord` function can be applied to values of type `point1`, and any subtype of `point1`, such as `cpoint1`. The `color` function, on the other hand, cannot be applied to values of type `point1`.

To make the notation a bit more "object-oriented," we introduce the following derived form for selecting a member:[3]

```
O$m ≡ (fn (O{|m, ...|}) => m)
```

For example, we can write a distance function on $\Re^2$ points as:

```
fun dist (p1, p2) = let
      val dx = (PT2$x p1)-(PT2$x p2)
      val dy = (PT2$y p1)-(PT2$y p2)
      in
        sqrt (dx*dx + dy*dy)
      end
```

Most accesses to members in programs use the derived form rather than full pattern matching.

Pattern matching on object constructors provides a form of typecase on objects, similar to that found in Modula-3 [Nel91]. For example consider the following expression:

```
case pt
 of CPT1{|...|} => print "color point\n"
  | PT1{|...|} => print "point\n"
```

If `pt` is a `cpoint1` (or some subtype of `cpoint1`), then the first case is matched. Otherwise the second case is matched (for example, if `pt` is a `point2`). Note that if the cases were in the other order, the second case (i.e., the `CPT1` case) would be redundant — this mistake can be detected by the compiler and reported as a warning. As we show later, we can use object pattern matching to compensate for the weakness of our type system.

3 Objects with methods

The examples so far have just used *fields*, which are the simple form of members. OML also provides *methods*, which are member functions that are implicitly passed the object as a parameter when they are invoked. For example, here is the interface to a point object (in $\Re^1$) that has three methods:

```
objtype point = PT of {|
    x : real,
    move : meth real -> point,
    bump : meth unit -> point,
    dist : meth real -> real
  |}
```

[3]This is similar to the #*l* notation for record field selection in SML.

The `move` method moves the point by its argument, the `bump` method moves the point by some fixed amount, and the `dist` method returns the distance between the point and its argument.

The following function creates a point from an initial x-coordinate:

```
fun mkPoint x0 = PT{|
    x = x0,
    move = meth self dx => mkPoint($x self + dx),
    bump = meth self () => $move self 1.0,
    dist = meth self x' => abs($x self - x')
  |}
```

We shorten the notation "`PT$x`" to just "`$x`" in the body of a method, where the object constructor `PT` is implicit. The methods are curried functions, with their first arguments being the object, and the second being the other arguments. Methods in OML have the so-called *self-application semantics* in which the first argument (i.e., `self`) becomes bound to the object during method invocation.

The `move` method adds `dx` to the `x` component of the object and then calls `mkPoint` to make a new point with new `x` value; the `bump` method uses `move` to move the point by one; and the `dist` function has the obvious implementation.

While the `move` and `bump` methods are applicative, it is also possible to define "stateful" objects by defining mutable fields, or by using local reference cells. For example,

```
fun mkPoint x0 = VARPT{|
    x = ref x0,
    move = meth self dx =>
             ($x self) := !($x self)+dx,
    bump = meth self () => $move self 1.0,
    dist = meth self x' => abs(!($x self) - x')
  |}
```

creates a mutable point, which is an implementation of the type

```
objtype varpt = VARPT of {|
    x : real ref,
    move : meth real -> unit,
    bump : meth unit -> unit,
    dist : meth real -> real
  |}
```

While many uses of OML objects will be imperative in practice, we focus on applicative objects because the requirements on the type system are more severe (imperative methods usually have a `unit` return type, which makes subtyping easier).

As before, we can define subtypes of an object type. Here is the type of color points, with a method for changing the object's color, and an implementation of color points:

```
objtype cpoint is point = CPT of {|
    c : color,
    shade : meth color -> cpoint
  |}

fun mkCPoint (x0, c0) = CPT{|
    x = x0, c = c0,
    move  = meth self dx =>
              mkCPoint($x self + dx, $c self),
    bump  = meth self () => $move self 1.0,
    dist  = meth self x' => abs($x self - x'),
    shade = meth self c' => mkCPoint($x self, c')
  |}
```

In this example, `CPT$move` has the type

```
cpoint -> real -> point
```

even though the value it returns is a `cpoint`, and so information is being lost. While it is possible to recover this information using pattern matching, a simple modification to the type system allows this information to be preserved statically. We introduce a special type constant, **`selfty`**, that can be used when specifying the types of object members. When typing an object constructor, we replace the occurrences of **`selfty`** with the constructor's object type. For example, if the `point` object type is defined as:

```
objtype point = PT of {|
    x : real,
    move : meth real -> selfty,
    bump : meth unit -> selfty,
    dist : meth real -> real
  |}
objtype cpoint is point = CPT of {|
    c : color,
    shade : meth color -> selfty
  |}
```

Then the type of `CPT$move` is:

```
cpoint -> real -> cpoint
```

Note that because the argument position of a function type is contravariant in the subtyping relation, **`selfty`** can only appear in positive positions.

4 Interactions with SML

When adding a new feature to a language, it is important to examine how it interacts with existing features. In general, our object types fit into SML in a natural fashion. For instance, type checking and compiling can proceed incrementally; it is not necessary to know the subtypes of an object type when compiling uses of the type, and hence interactive SML programming and separate compilation [AM94] are not affected by the addition of object types.

SML allows type definitions, such as datatype declarations, to be parameterized. The `list` type definition on page 2 is one example of such a type constructor. We would like to extend the subtyping relation to instances of such constructors; for example, we would like a list of color points to be a subtype of a list of points. Care must be taken, however, to avoid unsound subtyping. For example,

```
datatype 'a t = A of unit -> 'a | B of 'a -> unit
```

has its argument appearing in both positive and negative positions. Thus, a "`cpt t`" value is not comparable to a "`pt t`" under the subtyping relation. To handle this case, we must analyze the use of type arguments in the definition of type constructors, and must treat abstract type constructors conservatively.

It is also reasonable to parameterize **`objtype`** declarations to define object type constructors. We allow definitions of the form:

objtype ($\overrightarrow{\alpha}$) *tyid'* **is** ($\overrightarrow{ty}$) *tyid* **=** *conid* **of {|**
    lab_1 **:** ty_1**,** ..., lab_m **:** ty_m
|}

where the $\overrightarrow{\alpha}$ are the parameters of the object type constructor $tyid'$, and the $\overrightarrow{ty}$ are the instantiation of the parameters of the base type $tyid$. At the time of this writing, we have not had much experience with object type constructors, and are not sure of their utility.

For object types to work with the SML module system, we must introduce a specification form for object types in signatures. In addition to allowing a full object type definition in signatures, we also provide a *partially abstract* specification

objtype *ty* **is** *ty'*

which specifies that the object type *ty* is a subtype of *ty'*, but hides the representation of *ty* (i.e., what extra members it provides). For example, the signature

```
signature SIG = sig
    objtype a = A of {|x : int|}
    objtype b is a
  end
```

is matched by the structure

```
structure A : SIG = struct
    objtype a = A of {|x : int|}
    objtype b is a = B of {|y : int|}
  end
```

When using partially abstract object types in functor parameters, it is useful to be able to rename the object type. We support this by allowing an `objtype` specification in a signature to match a type definition, where the defined type is bound to an object type that matches the specification. For example, the following structure matches the `A` signature:

```
structure A : SIG = struct
    objtype a = A of {|x : int|}
    objtype c is a = C of {|y : int|}
    type b = c
  end
```

5 Object-oriented idioms

OML supports many of the standard object-oriented idioms found in the literature. Most important, it supports subtyping, which is the basis for many object-oriented programming techniques. In this section, we give examples of how we support some of the other techniques in OML.

5.1 Extensible datatypes

Object constructor matching provides a form of *extensible datatype*, in some ways, similar to SML `exn` type. In OML, we can replace the `exn` type and associated declarative form with a base type of exception objects. More specific exceptions can be defined as subtypes. For example, the arithmetic exceptions might be defined in the following way:

```
objtype exn = EXN of {| |}
objtype arith_exn is exn = ARITH of {| |}
objtype ovfl_exn is arith_exn = Overflow of {| |}
objtype div_exn is arith_exn = Div of {| |}
```

This allows an exception handler to catch collections of related exceptions, while letting others pass through. Similar mechanisms can be found in CLOS [Ste90], C++ [Str94], and Dylan [App92].

The extensible nature of object types is also useful for providing *open* implementations of heterogeneous collections. For example, we can define a base type of items:

```
objtype item = ITEM of {| |}
```

and implement operations over various collections of items. At any point that we want to put a new type of value into a collection, we can define a new subtype of the `item` type.

5.2 Binary methods

A *binary method* is a method whose argument has the same type as the object. While OML's type system is not rich enough to support true binary methods, because **`selfty`** cannot appear in the argument position of a method type, it is possible to use the object pattern matching to implement something close to binary methods. For example, here are points and color points with an `eq` method that tests for equality:

```
objtype point = PT {| x : int, eq : point -> bool |}
objtype cpoint is point = CPT {| c : color |}
fun mkPt x = PT{|
    x = x,
    eq = meth self pt => ($x self = PT$x pt)
  |}
fun mkCPt (x, c) = CPT{|
    x = x, c = c,
    eq = meth self pt => (case pt
            of (CPT{|x, c, ...|}) =>
                 (($x self = x)
                   andalso ($c self = c))
             | _ => ($x self = PT$x pt)
           (* end case *))
  |}

fun equal (p1, p2) = PT$eq p1 p2
```

Our type system gives `equal` the type:

```
(point * point) -> bool
```

Note that this technique can also be used to implement multi-methods — methods that dispatch on the type of more than one argument. In fact, the technique closely resembles Castagna's *encapsulated* multi-methods in which the definitions of multi-methods are local to the object [Cas95].

5.3 Friend functions

Another concept that is related to binary methods are C++ *friend* functions, which are functions that have access to the private members of an object [Str94]. Following Pierce and Turner [PT93b], we can use partially abstract object types to implement a form of friend functions. The approach is to define a base object type that contains the public operations, and a derived type that extends the public interface with private members. We group these definitions, along with the friend functions, in a module, and export the derived type as a partially abstract object type. For example, we might have the following interface to an implementation of sets of integers:

```
structure Set : sig

    objtype set = SET of {|
        add : meth int -> selfty,
        member : meth : int -> bool
      |}

    objtype set' is set

    val singleton : int -> set'
    val union     : (set' * set') -> set'

  end = struct ... end
```

The object type `set` defines the public interface, which consists of two operations: `add` for adding elements, and `member` for testing membership. The type `set'` is the representation type, but its additional members are not visible outside the `Set` module. The functions `singleton` and `union` have access to the internal representation type. The definition of the `set'` type might be

```
objtype set' is set = SET' of {|
    elems : int list
  |}
```

where `elems` is the list of set elements. The rest of the implementation follows from this.

6 Implementation inheritance

One important feature of many object-oriented languages is support for inheritance of implementations. Typically, this is supported either by a *delegation* mechanism or a *class* mechanism. Abadi and Cardelli's object calculi support delegation via object cloning and method override operations [AC95a]. We considered adding such operations to OML, but discovered that type safety imposes severe restrictions on their use in our setting.

Abadi and Cardelli also show how classes can be encoded as stylized objects, consisting of a `new` operation and *pre-methods*. We have been exploring encodings of classes

based on this approach. In OML, we can use objects to encode classes, but we can also use the SML module system. The latter seems more in keeping with the notion of a class, and it is what we present here.

In this encoding, a *class type* is an SML signature, and a *class* is a structure. For example, recall the `point` object type defined in Section 3; the corresponding class signature is:

```
signature PT_CLASS = sig
    val x : real
    val move : point -> real -> point
    val bump : point -> unit -> point
    val dist : point -> real -> real
    val new : unit -> point
  end
```

The `new` operation is used to construct new objects, while the pre-methods are used to inherit implementations. A class signature may have multiple implementations (structures). For example, here is an implementation of the `PT_CLASS` signature:

```
structure PTClass : PT_CLASS = struct
    val x = 0.0
    fun mk x0 = PT{|
        x = x0,
        move = meth ptSelf arg => move ptSelf arg,
        bump = meth ptSelf arg => bump ptSelf arg,
        dist = meth ptSelf arg => dist ptSelf arg
      |}
    and move pt dx = mk(PT$x pt + dx)
    and bump pt () = PT$move pt 1.0
    and dist pt x' = abs(PT$x pt - x')
    fun new () = mk x
  end
```

Note that the implementation of the `mk` function is uniform, and could be easily mechanized.

Inheritance of implementation is achieved by using the pre-methods of some existing class. For example, the `cpoint` object type from Section 3 has the class signature:

```
signature CPT_CLASS = sig
    val x : real
    val move : cpoint -> real -> cpoint
    val bump : cpoint -> unit -> cpoint
    val dist : cpoint -> real -> real
    val c : color
    val shade : cpoint -> color -> cpoint
    val new : unit -> cpoint
  end
```

An implementation of `CPT_CLASS` can inherit from an implementation of `PT_CLASS`:

```
structure CPTClass : CPT_CLASS = struct
  val x = PTClass.x
  val c = red
  fun mk (x0, c0) = CPT{|
      x = x0,
      move = meth ptSelf arg => move ptSelf arg,
      bump = meth ptSelf arg => bump ptSelf arg,
      dist = meth ptSelf arg => dist ptSelf arg,
      c = c0,
      shade = meth ptSelf arg => shade ptSelf arg
    |}
  and move cpt dx = mk (CPT$x cpt + dx, CPT$c cpt)
  and bump pt () = CPT$move pt 1.0
  and dist cpt x' = PTClass.dist cpt x'
  and shade cpt c' = mk (CPT$x cpt, c')
  fun new () = mk(x, c)
end
```

In this class, the definition of `x` and the implementation of `dist` are inherited from the `PTClass`. Note that there is no reason to fix the particular implementation of the point class that the color point class inherits from; we could define the color point class as a functor parameterized over the point class.

This example also illustrates a weakness of our type system. Abadi and Cardelli are able to encode the inheritance of methods that return **`selfty`**, such as `move` and `bump`, but this is not possible in OML. A method that returns `selfty` does so in one of two ways: it can return its self parameter, or it calls the `mk` function for the class to implement a functional update. It is this latter case that causes problems for the inheritance of method implementations. The obvious approach is to parameterize a class definition by the functional update operation, but there are some circularities that we have not been able to solve in a type correct manner. We are exploring the addition of a declarative mechanism for classes that would relieve the programmer of the tedious translation from object types to class interfaces and `new` operations. In addition, it might provide a framework for defining the inheritance of methods that return **`selfty`**.

7 Theoretical Properties of OML

A rigorous formal treatment of OML is outside the scope of this paper, but we describe some of the more important theoretical properties without proof.

We illustrate the formal properties via a language called OML--, a version of OML without reals, strings, pairs, lists, and other basic data types, without side effects and recursive functions, and whose objects involve only methods. The type rules are easier to state in this reduced language, and the extension to full OML follows the design of SML [MTH90]. The type rules appear in Table 1. Types, type schemes, constraint sets, and constraints are defined by the grammar

$$\begin{array}{rcl} s, t & ::= & \alpha \mid \textbf{selfty} \mid (s \to s) \mid obj \\ S, T & ::= & \forall\{\alpha_1 \ldots \alpha_n\} : C.\ s \\ C & ::= & \{A_1, \ldots, A_k\} \\ A & ::= & (\alpha \prec obj) \mid (obj \prec \alpha) \mid (obj \prec obj') \end{array}$$

Table 1: Typing Rules for OML--.

$$\frac{\Gamma(x) = \forall\{\alpha_1 \ldots \alpha_n\} : C'.\ s \quad C \vdash C'[t_1/\alpha_1, \ldots, t_n/\alpha_n]}{C; D; \Gamma \vdash x : s[t_1/\alpha_1, \ldots, t_n/\alpha_n]}$$

$$\frac{C; D; \Gamma \pm \{x \mapsto s\} \vdash M : t}{C; D; \Gamma \vdash (\mathbf{fn}\ x\ \mathbf{=>}\ M) : (s \to t)}$$

$$\frac{C; D; \Gamma \vdash M : (s \to t) \quad C; D; \Gamma \vdash N : s}{C; D; \Gamma \vdash (M\ N) : t}$$

$$\frac{C; D; \Gamma \vdash V : t \quad C; D; \Gamma \pm \{x \mapsto Gen(C, \Gamma, t)\} \vdash N : s}{C; D; \Gamma \vdash (\mathbf{let}\ x = V\ \mathbf{in}\ N) : s}$$

$$\frac{\begin{array}{c}(\texttt{objtype}\ obj = Obj\ \{|\ l_1 : (s_1 \to s_1'), \ldots, l_n : (s_n \to s_n')\ |\}) \in D \quad C \vdash obj \prec t \\ (\forall i.\ C \pm \{obj \prec \alpha\}; D; \Gamma \pm \{\texttt{self} \mapsto obj, x \mapsto s_i[\alpha/\mathbf{selfty}]\} \vdash M_i : s_i'[\alpha/\mathbf{selfty}])\end{array}}{C; D; \Gamma \vdash Obj\ \{|\ l_1 = \mathbf{meth}\ \texttt{self}\ x\ \mathbf{=>}\ M_1, \ldots, l_n = \mathbf{meth}\ \texttt{self}\ x\ \mathbf{=>}\ M_n\ |\} : t}\ \alpha \text{ fresh}$$

$$\frac{\begin{array}{c}C; D; \Gamma \vdash M : s \quad C \vdash s \prec s_n \quad C \vdash s_n = \text{least upper bound of } \{s_1, \ldots, s_n\} \\ (\forall i.\ s; D \vdash P_i : s_i, \Gamma_i \quad \text{and} \quad C; D; \Gamma \pm \Gamma_i \vdash M_i : t)\end{array}}{C; D; \Gamma \vdash (\mathbf{case}\ M\ \mathbf{of}\ \{\ P_1\mathbf{=>}\ M_1, \ldots, P_n\mathbf{=>}\ M_n\ \}) : t}$$

$$\frac{(\texttt{objtype}\ obj = Obj\ \{|\ l_1 : t_1, \ldots, l_n : t_n\ |\}) \in D}{s; D \vdash Obj\ \{|\ l_1, \ldots, l_n\ |\} : obj, \{l_i \mapsto t_i[s/\mathbf{selfty}] \mid 1 \le i \le n\}}$$

where α ranges over a predefined set of type variables and obj ranges over a class of object type identifiers. C denotes a set of subtyping assumptions among type variables and object types; the order of the subtyping assumptions A is unimportant, and we identify type schemes whose bound variables $\alpha_1 \ldots \alpha_n$ and subtyping assumptions C are the same up to reordering. Subtyping in the type system is completely captured by the following atomic subtyping rules:

$$C \vdash A, \text{ if } A \in C$$

$$\frac{C \vdash t \prec t' \quad C \vdash t' \prec t''}{C \vdash t \prec t''}$$

We write $C \vdash C'$ as shorthand for $C \vdash A$ for all $A \in C'$.

We will come back to the relationship between OML and OML-- in a moment, and first consider three theoretical properties of OML--. The two first technical result shows that complete, well-typed OML-- programs do not produce run-time type errors. Here, a **complete** OML-- program is a collection of object type declarations followed by a closed expression of any type. By "closed" we mean that there are no free variables, and any object constructor used in the expression has a corresponding declaration. The first property, called "subject reduction," states that well-typed expressions continue to be well-typed during evaluation:

Theorem 7.1 (Subject Reduction) *If the term M is closed and has type t, and M reduces to M', then M' has type t.*

The second property captures a dual notion: if a program is well-typed and has not returned a final answer, then its execution may continue:

Theorem 7.2 (Progress) *If the term M is closed and has type t, and M is not a value, then M can be reduced.*

Since a term with a run-time type error cannot be reduced under the reduction rules, both properties imply that well-typed programs do not cause run-time type errors.

The third technical result relates the typechecking rules to a type inference algorithm. The algorithm, a modification of the one found in [Mit84], itself relies on a modification of the standard unification algorithm: one finds most general unifiers subject to a set of *atomic subtyping constraints*. The algorithm finds a *principal type* (or *most general type*) for an expression.

The second technical result is the place where OML and OML-- diverge. Any type produced by the algorithm for OML is an instance of the type produced by the algorithm for OML--, but not conversely. We have made a fundamental tradeoff in the design of OML to simplify the type system: the algorithm for OML-- deduces types with bounded quantification, whereas the algorithm for OML uses simple subtyping. In fact, a rather old example due to Mitchell [Mit84] shows that type inference in OML cannot produce a principal type. Consider the function

```
fn p => case (p, PT$x p) of (x, _) => x
```

This expression has type $\forall\, \alpha \prec \texttt{pt}.\alpha \to \alpha$ in OML--, and so has both the type $\texttt{pt} \to \texttt{pt}$ (which is what OML assigns to it) and the type $\texttt{cpt} \to \texttt{cpt}$. Since these types are unrelated by the subtyping relation, neither is more general than the other. In our view, the former type is "better" than the latter, since it can be applied to a larger set of arguments, and we

can use pattern matching to recover the lost type information at run-time when necessary. It is also important to point out that while the OML typechecker does not produce a principal type (because such types do not exist in all cases), it does produce a "canonical" type. Specifically, it produces an instance of the most general type OML-- type, where bound type variables are instantiated to their bounds.

8 Implementation issues

We have been exploring these language ideas using a simple parser and typechecker that supports core-ML extended with our object types. This front-end produces a typed abstract syntax tree that we can interpret. We have used this implementation to test small examples, such as those found in this paper, and to explore the implementation issues in the OML typechecker. The implementation of a type-checker for OML requires fairly simple changes to the standard algorithm. The main difference is that the unification algorithm must compute the join or meet of two object types when unifying them. There is also some bookkeeping required to distinguish positive and negative contexts during unification.

An important property of OML is that it can be implemented efficiently without sophisticated compiler techniques. The main design issues are the representation of objects and the implementation of pattern matching on object constructors.

We define the *depth* of an object type to be the number of supertypes that it has. For each **`objtype`** declaration, we allocate a static *info structure* with the following **C** representation:

```
typedef struct objinfo {
    int            depth;
    struct objinfo *info[depth+1];
} objinfo_t, *objid_t;
```

where the allocated size of the `info` array is one greater than the object's depth. We use the address of this structure as the object type's unique ID. Each slot of the `info` array contains the ID of the object type's supertype at the corresponding depth. For each object value, we include a pointer to its info structure as its first word; this gives the following **C** representation:

```
typedef struct {
    objid_t  id;
    void     *members[1];
} object_t;
```

If `obj` is a pointer to an object, and O is an object constructor with unique ID `ID` and depth `D`, then we can match against O as follows:

```
if ((p->id->depth >= D)
&& (p->id->info[D] == ID)) {
    /* this matches the O object constructor */
}
```

For large collections of pattern matches, we may be able to use analysis of the object type hierarchy to reduce the number of comparisons.

A pattern match on an object type is *exhaustive* if it has a wild card, or specifically matches its argument type. In the case of a singleton exhaustive match, no dynamic checking is required, since the static typing ensures that the match will always succeed. In particular, the matches defined by the notation *O*$*m* are always exhaustive, and thus can be implemented as direct pointer dereferences, without conditionals.

One implementation issue that warrants further study is the space requirements of objects. Because an object type defines an interface that may have multiple implementations, our object representation must have slots for each member. This leads to a much fatter representation than found in class-based languages, such as C++, where the method functions are stored in a single table that is shared by all object instances. This problem is further exacerbated by the fact that functions in SML-like languages are represented as heap-allocated closures, and the methods for two different objects created by the same code may have different closure objects. One approach may be to adopt a *split* representation, where object methods and fields are stored in separate pieces of memory. Furthermore, the compiler may need to treat method functions specially to avoid the problems with the standard closure representation, although recent improvements in compiler techniques for SML may make this unnecessary [Sha94].

9 Related work

Over the past decade, many researchers have attempted to bridge the gap between the practice of dynamically typed object-oriented programming and the desire for static typechecking. This research has followed two different avenues: one avenue developed extensions of the λ-calculus that can encode objects (e.g., [Car84] and [Pie94]), and the other has developed calculi in which objects are a primitive notion (e.g., [Bru94] and [AC95a]). The most complete object-oriented type systems require resolving a fundamental conflict between subtyping (in which an object may be used in a context that expects a supertype of the object) and inheritance (in which definitions of objects may be reused to build new objects). Naive type systems that identify the two concepts may have problems with type soundness (see [Coo89] for an example in early versions of Eiffel [Mey92]). To avoid the problem, typed languages have used substantial type-theoretic machinery: higher-order subtyping [AC95b], bounded polymorphism [CW85], higher-order and F-bounded polymorphism [CCH$^+$89], existential types [PT93a], and matching [Bru94]. In many of these proposed systems, typechecking is not decidable.

The problems with typechecking magnify in the context of type inference if one keeps the same goals. Mitchell's seminal work on subtyping and type inference describes an algorithm and type rules for predeclared, fixed subtyping hierarchies [Mit84]; this system forms the basis of our OML-- calculus. Subsequently, this has been extended to

handle object-oriented features. The first extension is simple record subtyping: a record type with more fields is a subtype of a record type with fewer fields. Record subtyping requires a rather substantial change to the type inference system (for example, see [JM88], [Rém94], or [Wan87]). But record subtyping alone is not powerful enough to encode full objects: one requires some form of recursive type, since the types of methods should be able to return objects of the type itself (possibly through *self*). Existing type inference engines either solve a recursive set of constraints about types [Pal94], or collect a set of recursive constraints about types and report them to the programmer [EST95]. Neither solution is simple: it may require substantial insight on the part of the programmer to determine where a type error has occurred, as it is often not obvious when the recursive constraints have a solution.

Perhaps the closest type system to ours is the encapsulated multi-methods of [Cas95]. In this approach, methods exist locally within an object but can be overloaded. Our `case` construct yields a similar behavior. The main difference between the systems is in the type structure: our system does not have "overloaded" types.

The dynamic semantics of our objects is influenced by Abadi and Cardelli's object calculi [AC95a]. Their system supports delegation via object cloning and method override operations [AC95a]. While we might have added such operations to OML, the associated typing restrictions are complicated and it may be the case that our type system is not powerful enough to make such operations useful. They have also shown how classes can be encoded as a stylized objects; it is this approach that we follow in Section 6.

Thorup and Tofte have shown how to encode many object-oriented features in SML (including F-bounded quantification), but their encodings are quite cumbersome [TT94]. It would be interesting to see if their encodings are more direct in OML.

10 Conclusions and future work

We have presented a simple extension to SML that supports object-oriented programming. Our extension preserves the good properties of the SML type system — type soundness, polymorphism, type inference — while adding additional flexibility in the form of object subtyping. So far, we have been pleased with the expressiveness of OML; many natural examples are straightforward to program in the language, but we need more experience to fully evaluate the design. There may well be more facilities that need to be added to the language, such as some direct support for classes or more extensive object primitives.

We are incorporating these ideas into the design of MOBY, which is a small ML-like language for programming embedded systems. This will provide a better testament to the usefulness of our approach.

We are also exploring the relationship between the simple type system used by OML and the bounded types used by OML--. While the latter system has better formal properties, it is more complex and is harder to explain to programmers. We hope that studying the OML-- system will provide insight to the choices made by the OML type system.

Acknowledgements

We thank Kim Bruce, Kathleen Fisher, Jens Palsberg, Benjamin Pierce, Scott Smith, Valery Trifinov, and David Turner for helpful discussions, and Emden Gansner, Lal George, and Anne Rogers for reading drafts of this paper.

REFERENCES

[AC95a] Abadi, M. and L. Cardelli. An imperative object calculus. In *TAPSOFT'95: Theory and Practice of Software Development*, number 915 in Lecture Notes in Computer Science. Springer-Verlag, May 1995, pp. 471–485.

[AC95b] Abadi, M. and L. Cardelli. On subtyping and matching. In *ECOOP'95*, 1995.

[AM94] Appel, A. W. and D. B. MacQueen. Separate compilation for Standard ML. In *Conference Record of the 1994 ACM Conference on Programming Language Design and Implementation*, June 1994, pp. 13–23.

[App92] Apple Computer, Cambridge, MA. *Dylan: An Object-oriented Dynamic Language*, 1992.

[Bru94] Bruce, K. B. A paradigmatic object-oriented language: design, static typing, and semantics. *Journal of Functional Programming*, **4**(2), 1994, pp. 127–206.

[Car84] Cardelli, L. A semantics of multiple inheritance. In *Semantics of Data Types*, vol. 173 of *Lecture Notes in Computer Science*, New York, N.Y., 1984. Springer-Verlag, pp. 51–67.

[Car95] Cardelli, L. A language with distributed scope. In *Conference Record of the 22th Annual ACM Symposium on Principles of Programming Languages*, January 1995, pp. 286–297.

[Cas95] Castagna, G. Covariance versus contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, **17**(3), May 1995, pp. 431–447.

[CCH+89] Canning, P., W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming. In *Conference Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. ACM, 1989, pp. 273–280.

[Coo89] Cook, W. R. A proposal for making Eiffel type-safe. In *European Conference on Object-Oriented Programming*, 1989, pp. 57–72.

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

[DM82] Damas, L. and R. Milner. Principal types for functional programs. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, January 1982, pp. 207–212.

[EST95] Eifrig, J., S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *OOPSLA'95 Proceedings*. ACM, 1995. To appear.

[FM94] Fisher, K. and J. C. Mitchell. Notes on typed object-oriented programming. In *TACS'94: Theoretical Aspects of Computer Science*, number 789 in Lecture Notes in Computer Science. Springer-Verlag, 1994, pp. 844–885.

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

[GR93] Gansner, E. R. and J. H. Reppy. *A Multi-threaded Higher-order User Interface Toolkit*, vol. 1 of *Software Trends*, pp. 61–80. John Wiley & Sons, 1993.

[JM88] Jategaonkar, L. and J. C. Mitchell. ML with extended pattern matching and subtypes. In *Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, July 1988, pp. 198–211.

[Mey92] Meyer, B. *Eiffel: The Language*. Prentice Hall, New York, NY, 1992.

[Mit84] Mitchell, J. C. Coercion and type inference (summary). In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, January 1984, pp. 175–185.

[MTH90] Milner, R., M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass, 1990.

[Nel91] Nelson, G. (ed.). *Systems Programming with Modula-3*. Prentice-Hall, Englewood Cliffs, N.J., 1991.

[Pal94] Palsberg, J. Efficient inference of object types. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, 1994, pp. 186–195.

[Pie94] Pierce, B. Bounded quantification is undecidable. In C. A. Gunter and J. C. Mitchell (eds.), *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*, pp. 427–459. MIT Press, 1994.

[PT93a] Pierce, B. C. and D. N. Turner. Object-oriented programming without recursive types. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, January 1993, pp. 299–312.

[PT93b] Pierce, B. C. and D. N. Turner. Statically typed friendly functions via partially abstract types. *Technical Report ECS-LFCS-93-256*, University of Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[Rém94] Rémy, D. Type inference for records in a natural extension of ML. In C. A. Gunter and J. C. Mitchell (eds.), *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*, pp. 67–95. MIT Press, 1994. Also available as Research Report 1431, May 1991, INRIA-Rocquencourt, France.

[Sha94] Shao, Z. *Compiling Standard ML for Efficient Execution on Modern Machines.* Ph.D. dissertation, Princeton University, Department of Computer Science, November 1994.

[Sol78] Solomon, M. Type definitions with parameters. In *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages*, January 1978, pp. 31–38.

[Ste90] Steele, Jr., G. L. *Common Lisp: The Language*. Digital Press, 2nd edition, 1990.

[Str94] Stroustrup, B. *The Design and Evolution of C++*. Addison Wesley, Reading, Mass., 1994.

[TT94] Thorup, L. and M. Tofte. Object-oriented programming and Standard ML. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, June 1994, pp. 41–49. Available as INRIA Research Report No. 2265.

[US87] Ungar, D. and R. B. Smith. Self: The power of simplicity. In *OOPSLA'87 Proceedings*, October 1987, pp. 227–242.

[Wan87] Wand, M. Complete type inference for simple objects. In *Proceedings, Symposium on Logic in Computer Science*. IEEE, 1987, pp. 37–44. Corrigendum in *Proceedings, Symposium on Logic in Computer Science*, page 132. IEEE, 1988.