Objects in a Functional Environment

Juan Carlos Guzmán and Ascánder Suárez

Universidad Simón Bolívar – Departamento de Computación

Valle de Sartenejas, Caracas 1080, Venezuela

Email: {jcguzman,suarez}@usb.ve

Abstract

The Object Oriented Paradigm of Programming has a great impact in the way programs and systems are designed today. Nevertheless, its influence in the Functional Programming community is quite limited. No functional programming environment with objects is considered today as a standard, and new proposals continue to emerge. In this paper we study the issues concerning the functional manipulation of objects and propose new primitives that incorporate objects as first class elements of functional languages, adapt them to the pattern-matching mechanism and rend them compatible with the polymorphic type inference mechanism.

1 Introduction

The functional language community has been actively searching for a way of integrating objects within the functional framework. By now a great deal of research has been done

on topics such as ad-hoc polymorphism, inclusion polymorphism, subtyping, inheritance, abstract data types, etc. that has set the framework for such a merge. Further, some attempts have been made at integrating all these aspects into a prototype of ML [7, 9, 5] with objects [3, 8] but, as of today, this is regarded as an open problem [1].

In this paper, we survey the principal aspects present in languages with objects and discuss particular problems that pose the use of objects in a functional programming environment; In particular, the relation between polymorphism and objects and the definition of functions that map objects to objects. We continue with a methodic enumeration of the concepts tied to the object oriented style of programming and discuss their fitness in the functional programming environment.

2 Objects

The definition of the type list in the ML programming language reads as follows:

$$\textbf{\textit{type}}\ (\alpha)list = Nil \mid Cons\ \textbf{\textit{of}}\ (\alpha \times (\alpha)list)$$

In this construction, a new type "*list*" and two constructors, "*Nil*" and "*Cons*" are defined. A natural way to introduce objects seems to be the extension of presently available data structures. Thus, the type will become a class with different constructors. Methods will be included in definitions in a special clause associated to class constructors. A definition of lists in the new style will become:

$$\textbf{\textit{class}}\ List; \quad List\ \textbf{\textit{class}}\ Nil; \quad List\ \textbf{\textit{class}}\ (\alpha)Cons\ \textbf{\textit{of}}\ (\alpha \times (\alpha)list)\ \textbf{\textit{methods}}\ \ldots;$$

An object is constructed simply by showing it in a program. For instance, "*Nil*," "$Cons(2, Nil)$" and "$Cons(true, Nil)$" are expressions that evaluate in list objects. Simple inheritance is

obtained by extending a class constructor with new fields an methods. For instance, the constructor *Bond* could be defined as an extension of *Cons* with an extra generic field as follows:

$$Cons\ \textbf{class}\ (\beta)Bond\ \textbf{of}\ \beta;$$

In order to build an object with the *Bond* class constructor, it is necessary to give its own fields as well as the fields of the constructor it extend: $Bond(1.1)(true, Nil)$.

3 Objects as first class citizens

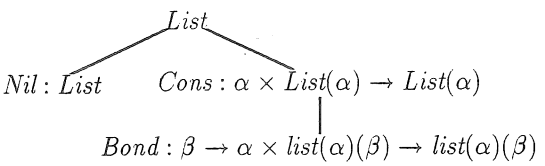
The object oriented programming community has developed the technology that is used in order to take advantage of objects in algorithmic languages. These issues are reviewed in Section 4 and their inclusion in a functional language is discussed. Nevertheless, the concept of functional objects, i.e. objects that can be manipulated like any other functional construct, is particular to this approach. We discuss three of this aspects in this section: The relation between classes and type inference in the Hindley-Milner style, the definition of functions that map objects to objects and their relation to inheritance, and, finally, the pattern-matching problem for objects.

3.1 Type inference

Modern functional languages take advantage of the type inference systems proposed by Hindley [4] and Milner [6, 2]. In this system, the type of expressions is reconstructed from the type of arguments and constants, and when possible, generic types are produced that then be instantiated to other more particular types. Objects, on the other hand, introduce a genericity based on inheritance. Roughly, a method defined for a class is known by its sub-classes,

and thus, objects of the sub-classes behave in a similar way with respect to the method.

A simple way to extend the Hindley-Milner type inference system starts by associating a type to a class hierarchy. For the example of lists in the preceding section, objects built from constructors *Nil*, *Cons* and *Bond* are all of a type named *List*. Different levels of genericity were introduced on the example for the definition of the sub-classes. This genericity is included as a parameter of the type in order to achieve type inference for data structures. Therefore, the type of *Cons* should have one parameter and the type of *Bond* should have two. The resulting types are the instances obtained by replacing α and β by types in $List(\alpha)$ and $List(\alpha)(\beta)$. It is desirable to manipulate objects as values of their super-classes, and thus, the type system should accept for instance, an object with type $List(Int)(Bool)$ as if it were an object with type $List(Int)$. The type system obtained allows inference of types for objects as in a very similar way to the usual Hindley-Milner type system. The class hierarchy corresponding to lists is



3.2 Functional mapping of objects

In algorithmic languages, it is common to send messages to objects in order for them to modify their state. In contrast, in functional languages, the usage is to obtain as a result, a new object with the modified state, letting compilers do the sharing when possible. For

instance, the function *map* defined as:

$$\textbf{rec } map = \textbf{fun } f\ Nil \rightarrow Nil$$
$$\mid\ f\ Cons(x, xs) \rightarrow Cons(f\ x, map\ f\ xs)$$

applys a function to all the elements of a list, producing a new list with the results obtained. When applying the function "(***fun*** $x \rightarrow x + 1$)" to the list "$Cons(1, Cons(2, Nil))$" we obtain the result "$Cons(2, Cons(3, Nil))$." Now that lists are objects, it should be possible to map functions built with extensions of *Cons*. Unfortunately, unless specific tools are provided, information concerning the sub-class will be lost:

$$map\ (\textbf{fun } x \rightarrow x + 1)\ (Bond(true)(1, Bond(false)(2, Nil))) = Cons(2, Cons(3, Nil)).$$

A better behavior for *map* should be to produce a list built with *Bond*'s and not with *Cons*'es. Nevertheless, the definition of *map* could have been made in a library and the extension of *Cons* in a program using that library, so it is necessary to provide a general mechanism to allow the use the constructor of an object as a method. Thus, $Bond(2)$ should be seen as a constructor of $List(\alpha)$ as well as *Cons* is.

The proposed extension is to integrate with the mechanism that allows to see an object as a value of a super-class, the binding of the constructor of the class partially applied, so that it behaves like the constructor of the super-class. In the *map* example, the construction of the resulting list should be done with the same constructor that was used to build its argument. This is expressed in a special case of pattern-matching, in which the constructor of the class is given a name. The definition of *map* with the constructor of *Cons*'es named c behaves as proposed.

$$\textbf{rec } map = \textbf{fun } f\ Nil \rightarrow Nil$$
$$\mid\ f\ c\_Cons(x, xs) \rightarrow c\_Cons(f\ x, map\ f\ xs)$$

3.3 Pattern-Matching

Call by pattern-matching is a very useful tool in functional programming. It allows to specify functions by cases on the structure of arguments and, at the same time, to bind variables to elements of them. The natural extension of pattern-matching for the inclusion of objects is to allow the use of constructors as patterns that match objects of its class and its sub-classes. Pattern cases can be ordered by priority, giving the possibility to write elegant selectors for functions over different extensions of a class.

As an example, let us take the following function definition and the evaluation of three calls to function f:

$$\textbf{\textit{let}}\ f = \textbf{\textit{fun}}\ (Cons(2, l)) \rightarrow 1 \mid (Bond(true)(x, Nil)) \rightarrow 2 \mid y \rightarrow 3;$$

$$f\,(Bond(true)(2, Nil)) = 1;$$

$$f\,(Bond(true)(3, Nil)) = 2;$$

$$f\,(Cons(3, Nil)) = 3;$$

Object $Bond(true)(2, Nil)$ is an instance of the first pattern (in fact of all of them, but the first one has the highest priority) as every *Bond* is a *Cons*. Object $Bond(true)(3, Nil)$ is an instance of the second pattern. Pattern x works as a wild-card, all values which are not instances of the first two are instances of the last one.

Two important constructs of object-oriented programming are also absorbed into the pattern-matching mechanism. They are coercions between compatible classes and selection of methods.

4 The Object-Oriented paradigm

Within the object-oriented programming paradigm, programs are viewed as collections of independent agents called *objects*. Computation proceeds by interaction among these agents.

Objects are classified in *classes*. A *class* is the type of the object. On the contrary, an object is an *instance* of a class. Distinguishing among different kinds of objects is important for establishing the behavior of different objects, as their behavior is determined by their class. The majority of object-oriented languages have the notion of static typing.

Each object knows how to handle a group of tasks. The algorithm used to solve each task (*method*) is specified in the object's class. Computation is attained by sending a *message* to an object to perform a specific task. A message is an instance of a problem to be solved by the object. There is also a state local to each object, as specified in its class. In functional programming, side-effects are outlawed, therefore, a new object needs to be constructed whenever a change in the local state is needed. Typing an object's method is not possible in the Hindley-Milner type system, however, we have devised a simple way of extending the type system so that these types can be inferred.

Classes are organized in hierarchies. Classes can be specialized, so data and methods defined there can be used (without redefinition) in the *derived* class. This mechanism of software reutilization is called *inheritance* and is very powerful. Derived classes can also define new methods, as well as provide new definitions for those inherited. In fact, it is fairly common for inherited methods to be somehow specialized, thereby making it possible for a derived class to specify only the computation proper to the class, and to refer to its parent's methods whenever possible. The Hindley-Milner type system does not support the notion of subtyping, but, again, this can be solved.

There are several choices to be made when dealing with inheritance. First, the hierarchy graph can really be a tree if every derived class has only one parent class. There is single inheritance in this case. Otherwise, a class can be derived from several others. This is known as multiple inheritance. Although the latter is more general, the implications of multiple inheritance are not completely understood yet.

Another issue regarding inheritance is the introduction of polymorphism in objects: the dynamic (run time) class of an object can be a derived class of the class computed at compile time. This matters when the derived class either defined or redefined a method—i.e., when the methods of both classes differ. The decision of which method to use affects the expressiveness and efficiency of the language. Functional programming is compatible with both approaches, and we have chosen to dynamicly resolve the methods.

Program structuring with objects encapsulates in each of these agents' local state plus methods. As mentioned earlier, as a result of the absence of assignment, a new object needs to be created—and returned—whenever a change of state is needed.

Other aspects corresponding to functional languages should be addressed: higher-order functions and pattern-matching.

5 An Example

We present in this section a more elaborated example where the reader can appreciate how we cast the object-oriented paradigm within the functional model of computation. In this example, we provide an object-oriented implementation of the 'call-by value' graph-reduction semantics for pure λ-calculus.

The technique used is to create an abstract class AST from which all AST's will be derived.

We then create class constructors *Var*, *Abs*, and *App*. These implement the specialized *AST* nodes for variables, abstractions, and applications.

The methods *eval*, and *subst* are defined for the three classes. The former evaluates an expression, the latter performs the substitution of a variable for an expression in an object:

$$exp.eval \longrightarrow exp' \qquad \textbf{if } exp \rightarrow exp' \textbf{ and } exp' \text{ is in 'weak head-normal-form'}$$

$$exp.substfvarexp' \longrightarrow exp[exp'\backslash fvar]$$

In addition, class *Abs* has the extra method *reduce*, which essentially handles the effect of the β reduction on an abstraction.

$$(\lambda bvar.body).reduceexp \longrightarrow body[exp\backslash bvar]$$

class *AST*

AST ***class*** *Var* ***of*** { *name*: *string* }

methods *eval* = *self*

and *subst fvar exp* = ***if*** *fvar*=*name* ***then*** *exp* ***else*** *self*;;

AST ***class*** *Abs* ***of*** { *bvar*: *string*; *body*: *AST* }

methods *eval* = *self*

and *subst fvar exp* = ***if*** *fvar*=*bvar* ***then*** *self* ***else*** *Abs bvar*=*self.bvar*; *body* =

subst fvar self.body ***and*** *reduce exp* = *body.subst fvar exp*;;

AST ***class*** *App* { *fnc*: *AST*; *arg*: *AST* }

methods *eval* = (*fnc.eval.reduce arg.eval*

and *subst fvar exp* = *App* { *fnc* = *self.fnc.subst fvar exp*; *arg* = *self.arg.subst*

fvar exp };;

Adding to the interpreter is now easy. We will extend it in two ways: by adding syntax,

such as *let* which extends node types to the definition of *AST*s, and also by adding the capability of operating with primitive functions.

Following is the addition of the new construct *let*. Since the semantics of a let-expressions combines those of an abstraction and an application, this derived class handles all three methods defined.

```
AST class Let { bvar: string; binding: AST; body: AST }
 methods eval = let bvalue = binding.eval in (body.subst bvar bvalue).eval
 and subst fvar exp = if fvar=bvar then self
     else Let { bvar =self.bvar; binding=self.binding.subst fvar exp;
                body =self.body.subst fvar exp }
```

Next is the addition of numbers as constants, primitive operation nodes, such as arithmetic operations, and the definition of a primitive operation.

```
AST class Primitive { name: string; fn: AST→AST }
 methods eval = self
 and subst fvar exp = self
 and reduce exp = fn exp;;

AST class Int { value: int }
 methods eval = self .
 and subst fvar exp = self;;
let plus (Int x) =
     Primitive { name=""; fn=fun (Int y) → Int value=x.value+y.value }
```

6 Conclusions

We have surveyed the most important issues in object-oriented languages, and have presented via examples a framework which allows the object model to coexist with the functional paradigm. Of particular importance are the extension of pattern-matching to handle classes and subclasses, and the typing of methods that correctly handles the type of the *self* argument. We study the relation between polymorphism and objects, as well as other issues relevant to the inclusion of the object-oriented paradigm in the functional programming model.

References

[1] A. Black and J. Palsberg. Foundations of object-oriented languages. *ACM SIGPLAN Notices*, 29(3):3–11, 3 1994.

[2] L. Damas and R. Milner. Principal type schemes for functional languages. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212. August 1982.

[3] D. Duggan. Object interfaces, polymorphic methods and multi-method dispatching for ml-like languages. In *Proceedings of the 1994 Workshop on ML and its Applications*, pages 50–61, Inria, B.P. 105, 78153 Le Chesnay Cedex, France, 06 1994.

[4] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[5] X. Leroy. The caml light system, release 0.6. Technical report, INRIA, 1993. Included with the Caml Light 0.6 Distribution.

[6] R. Milner. A theory of type polymorphism. *J. Computer and Systems Sciences*, 17:348–375, 1978.

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

[8] L. Thorup and M. Tofte. Object-oriented programming and standard ml. In *Proceedings of the 1994 Workshop on ML and its Applications*, pages 41–49, Inria, B.P. 105, 78153 Le Chesnay Cedex, France, 06 1994.

[9] P. Weis, M. Aponte, A. Laville, M. Mauny, and A. Suárez. The CAML reference manual. Technical report 121, INRIA, 9 1990.