Using modules as classes

Jérôme Vouillon*

November 9, 1997

Abstract

We show how classes can be combined with a ML-like module system, so that the same constructions be used for both classes and modules. For that, we are brought to slightly extend the module system with implicit type quantification of module types.

Introduction

Classes and modules shares a lot of properties: they offer some form of abstraction; they structure the code and facilitate its reuse. Due to this overlapping of functionalities, it is sometimes difficult in a language providing both constructions to make a choice between them. This paper attempts to reduce the gap between these two styles of programming by merging classes and modules: we show how the classes of Objective ML [5] can be encoded as modules. This requires a small but natural extension to existing ML-like module systems: just as ML types are implicitly quantified, we propose that module types be also implicitly quantified over some type variables. We of course also make some specific additions to the module language, such as a special form `new` for creating objects and a structure component `method`.

This paper is organized as follows. We first give a brief introduction to Objective ML[1]. In section 2, Objective ML classes are compared to modules, and we show how they can be combined. In section 3, we presents more formally the resulting language. In section 4, we discuss the impact of the proposed extension to modules. In section 5, we discuss type inference. In section 6, we explains what restrictions should be made to ensure soundness in presence of imperative features. In section 7, we show the limits of our integration of classes and modules. In section 8, we compare our proposal with other works.

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

[1]The reader is directed to [5] for a more complete description of Objective ML.

1 Classes in Objective ML: an overview

A simple class can be defined as a structure containing some instance variables and some methods.

$$\texttt{class}\ \mathit{origin} = \texttt{struct}\ \texttt{val}\ x = 0;\ \texttt{method}\ \mathit{getx} = \texttt{fun}\ y \to x\ \texttt{end}$$

This class has an instance variable x initialized to 0 and a method *getx*. When this method is invoked, y is bound to self (that is, the object the method belongs to.) In this case, this argument is actually ignored, and the method returns the value 0 of the instance variable. The type inferred for this class is

$$\forall(\alpha)\ \texttt{sig}\ (\langle\, \mathit{getx} : \mathit{int};\ \alpha\,\rangle)\ \texttt{val}\ x : \mathit{int};\ \texttt{method}\ \mathit{getx}\ \texttt{end}$$

The type between parentheses is the self type, the type of an object of this class. It is composed of a sequence of method types. The *row variable* α can be instantiated in subclasses to extend the class with new methods. The other fields of the signature reveal that the class contains an instance variable x and that the method *getx* is indeed defined.

Objective ML actually provides a simple class language. A class can be defined as a function taking an initialization parameter and returning a class value. This is demonstrated by the following class.

$$\texttt{class}\ \mathit{genpoint} = \texttt{fun}\ x_0 \to \texttt{struct}\ \texttt{val}\ x = x_0;\ \texttt{method}\ \mathit{getx} = \texttt{fun}\ y \to x\ \texttt{end}$$

The type of this class is

$$\forall(\alpha)\ \forall(\alpha')\ \big(\alpha \to \texttt{sig}\ (\langle\, \mathit{getx} : \alpha;\ \alpha'\,\rangle)\ \texttt{val}\ x : \alpha;\ \texttt{method}\ \mathit{getx}\ \texttt{end}\big)$$

Then, the class expression $\big(\mathit{genpoint}\ 1\big)$ is a class whose instance variable x is bound to 1. The operator `new` is used to create objects. An object is composed of a copy of the instance variables and methods of its class, but only its methods can be directly accessed. This is reflected by its type. For instance, $\big(\texttt{new}\ \mathit{genpoint}\ 1\big)$ is an object of type $\langle\, \mathit{getx} : \mathit{int};\ \epsilon\,\rangle$. In this type, the row variable has been instantiated to ϵ, as the object has no other method than *getx*.

Classes can be extended by inheritance. The component `inherit` c of a class structure includes all components of the parent structure c into the current structure. The type for self must be an instance of the parent type for self. The class *cpoint* below inherits the instance variable x and the method *getx* from its parent *genpoint*, and defines a new method *color*.

$$\begin{array}{l}\texttt{class}\ \mathit{cpoint} = \texttt{fun}\ (c : \mathit{string}) \to \\ \qquad\qquad\qquad \texttt{struct}\ \texttt{inherit}\ \mathit{genpoint}\ 1;\ \texttt{method}\ \mathit{color} = c\ \texttt{end}\end{array}$$

The type of class *cpoint*, given below, reveals the inherited instance variable and both methods.

$$\begin{array}{l}\forall(\alpha)\ \mathit{string} \to \\ \qquad \texttt{sig}\ (\langle\, \mathit{getx} : \mathit{int};\ \mathit{color} : \mathit{string};\ \alpha\,\rangle) \\ \qquad\quad \texttt{val}\ x : \mathit{int};\ \texttt{method}\ \mathit{getx};\ \texttt{method}\ \mathit{color} \\ \qquad \texttt{end}\end{array}$$

2 Comparison with a module system

As we have seen, Objective ML provides a small class language, with abstraction, application and structures. This is similar to a module language *a la* ML. Figure 1 provides an informal comparison of the syntax of these two languages, emphasizing their similarities. There are, however, some differences, and the remainder of this section will show how we can get rid of them.

A first difference is that a class abstraction maps values to class values, whereas a functor maps modules to modules. This could almost be overcome by encoding a class abstraction $\texttt{fun}\ x \rightarrow m$ as $\texttt{functor}\ (z : \texttt{sig val}\ x : \tau\ \texttt{end}) \rightarrow m[z.x/x]$, but the type τ is usually unknown. We will come back to this problem later.

A more serious problem is that class types are quantified and class structure components may have non-closed types: free type variables can occur in self type and in instance variable types. On the contrary, all core types occurring in a module type must be closed (and so, module types do not need to be quantified.)

It is actually possible to encode explicit quantification in the module language [2]: $\forall(\alpha)\ m$ can be rewritten as $\texttt{functor}\ (z : \texttt{sig type}\ t\ \texttt{end}) \rightarrow m[z.t/\alpha]$. However, the value of type t will have to be explicitly provided to remove the quantification. This does not mix well with a core language with implicit polymorphism such as ML.

We therefore propose to allow the implicit quantification of module types, using the following (natural) rules to introduce and eliminate quantification:

$$\frac{A \vdash m : \forall(\alpha)\ \Phi}{A \vdash m : \Phi[\tau/\alpha]} \qquad\qquad \frac{A \vdash m : \Phi \qquad \alpha \notin \mathrm{FV}(A)}{A \vdash m : \forall(\alpha)\ \Phi}$$

It appears useful to limit the implicit generalization of types variables at the top of structures. This can be done by explicitly binding type variables:

$$\frac{A; \big(\texttt{var}\ \alpha = \tau\big) \vdash m : \Phi}{A \vdash \big([\alpha]\ m\big) : \Phi}$$

During the typing of the module expression m, the type variable α is bound to some type τ. Then, the free variables of τ cannot be generalized in m, as they occur in the environment. With this extension, the expression $\big(\texttt{fun}\ x \rightarrow m\big)$ can be encoded as $[\alpha]\ \texttt{functor}\ (z : \texttt{sig val}\ x : \alpha\ \texttt{end}) \rightarrow m[z.x/x]$.

Other differences are all minor. Self type can be considered as just another component of a signature. Instance variables and value components have the same semantics. The clauses `inherit` and `open` are really equivalent: they both include the components from another structure into the current structure. They can be typed using the following typing rules (the notation $\phi +_A \phi'$ stands for the signature which holds the components of both ϕ and ϕ', and whose self type is the same as the ones of ϕ and ϕ'.)

Class	Module	
Class and module expressions		
$c ::= z$	$m ::= p$	Access to a named value
$\mid$ `fun` $z \to c$	$\mid$ `functor` $(z : \phi) \to m$	Abstraction
$\mid c\ e$	$\mid m\ m$	Application
$\mid$ `struct` b `end`	$\mid$ `struct` b `end`	Structure
$b ::= \epsilon \mid d;\ b$	$b ::= \epsilon \mid d;\ b$	
Structure components		
$d ::=$ `inherit` c	$d ::=$ `open` m	Inheritance / inclusion
$\mid$ `val` $x = e$	$\mid$ `val` $x = e$	Value
$\mid$ `method` $y =$ `fun` $x \to e$		Method
	$\mid$ `type` $t = \tau$	Type
	$\mid$ `module` $z = m$	Module
	$T ::= \tau \mid \forall(\alpha)\ T$	Type scheme
$\Phi ::= \phi \mid \forall(\alpha)\ \Phi$		Class type scheme
Class and module types		
$\phi ::= \tau \to \phi$	$\phi ::=$ `functor` $(z : \phi) \to \phi$	Functional type
$\mid$ `sig` $(\tau)\ \chi$ `end`	$\mid$ `sig` χ `end`	Signature type
$\chi ::= \epsilon \mid \psi;\ \chi$	$\chi ::= \epsilon \mid \psi;\ \chi$	
Signature components		
$\psi ::=$ `val` $x : \tau$	$\psi ::=$ `val` $x : T$	Value
$\mid$ `method` m		Method
	$\mid$ `type` $t = \tau$	Manifest type
	$\mid$ `type` t	Abstract type
	$\mid$ `module` $z : \phi$	Module

Figure 1: Comparison between a class language and a module language

$$\frac{A \vdash m : \phi}{A \vdash \big(\texttt{inherit}\, m\big) : \phi} \qquad\qquad \frac{A \vdash d : \phi \qquad A + \phi \vdash b : \phi'}{A \vdash \big(d;\, b\big) : \big(\phi +_A \phi'\big)}$$

Method components can be easily added, with the typing rule below. A method is typed like a function; the argument type of this function must be self type τ, while its result type must be the type of method y in τ.

$$\frac{A \vdash \big(\texttt{fun}\, x \to e\big) : \tau' \to \tau \qquad A \vdash \tau' \approx \langle\, y : \tau ;\, \omega \,\rangle}{A \vdash \big(\texttt{method}\, y = \texttt{fun}\, x \to e\big) : \big(\texttt{sig}\ (\tau')\ \texttt{method}\, y\ \texttt{end}\big)}$$

Finally, the typing rule for the operator `new` can be straightforwardly adapted from Objective ML: the type of the object $\texttt{new}\, m$ is the self type τ, and all its methods must be defined (rightmost premise.)

$$\frac{A \vdash m : \big(\texttt{sig}\ (\tau)\ \chi\ \texttt{end}\big) \qquad A \vdash \tau \approx \langle\, y : \tau_y \,\rangle^{(\texttt{method}\ y) \in \chi}}{A \vdash \big(\texttt{new}\, m\big) : \tau}$$

3 Presentation of the extended module system

We now describe the full module system. The module system presented here is a variant of the one proposed by Xavier Leroy [4]. In this variant, names appearing in a program are not annotated by stamps. Stamps only appears in types, where they are really necessary to resolve possible ambiguities. The core language is left mostly unspecified.

3.1 Syntax

The syntax is presented in figure 2. It is standard. z, t, x and y are names (for module, type, value and method components of structures, respectively.) Names are not subject to α-conversion. Types are denoted by Greek letters. Type schemes are denoted by the capital letter associated to the small letter denoting the corresponding types. The tildes ($\tilde{\ }$) over these syntactic types differentiate them from the enriched types that will be necessary for typing.

The following additions have been made to the syntax. Types are enriched with recursive types and object types. The module expression $[\alpha]\ m$ explicitly binds the type variable α in m. The component $\texttt{method}\, y = \texttt{fun}\, x \to e$ defines a method; during an invocation of this method, its body e is evaluated with the variable x bound to the object the method belongs to. The component $\texttt{inherit}\, m$ includes the components of the structure m into the current structure. Module types are quantified. A signature $\big(\texttt{sig}\ (\tilde{\tau})\ \tilde{\chi}\ \texttt{end}\big)$ is composed not only of a sequence of component types $\tilde{\chi}$, but also contains the self type $\tilde{\tau}$. The signature component $\texttt{method}\ y$ indicates that the module defines the method y.

$\tilde{p} ::= z \mid \tilde{p}.z$	Access path
$e ::= x \mid \tilde{p}.x \mid \texttt{new}\ m \mid \texttt{fun}\ x \to e \mid \ldots$	Expression
$\tilde{T} ::= \tilde{\tau} \mid \forall(\alpha)\ \tilde{T}$	Core type scheme
$\tilde{\tau} ::= t \mid \tilde{p}.t \mid \alpha \mid \tilde{\tau} \to \tilde{\tau} \mid \mu(\alpha)\ \tilde{\tau} \mid \langle\, \tilde{\omega}\, \rangle$	Core type
$\tilde{\omega} ::= \epsilon \mid \alpha \mid y : \tilde{\tau};\ \tilde{\omega}$	Object type
$m ::= \tilde{p} \mid \texttt{struct}\ b\ \texttt{end} \mid \texttt{functor}\ (z : \tilde{\Phi}) \to m \mid m\ m$ $\mid (m : \tilde{\Phi}) \mid [\alpha]\ m$	Module expression
$b ::= \epsilon \mid d;\ b$	Module body
$d ::= \texttt{val}\ x = e \mid \texttt{method}\ y = \texttt{fun}\ x \to e$ $\mid \texttt{module}\ z = m \mid \texttt{type}\ t = \tilde{\tau} \mid \texttt{inherit}\ m$	Module component
$\tilde{\Phi} ::= \tilde{\phi} \mid \forall(\alpha)\ \tilde{\Phi}$	Module type scheme
$\tilde{\phi} ::= \texttt{functor}\ (z : \tilde{\Phi}) \to \tilde{\Phi} \mid \texttt{sig}\ (\tilde{\tau})\ \tilde{\chi}\ \texttt{end}$	Module type
$\tilde{\chi} ::= \epsilon \mid \tilde{\psi};\ \tilde{\chi}$	Module body type
$\tilde{\psi} ::= \texttt{val}\ x : \tilde{T} \mid \texttt{method}\ y \mid \texttt{module}\ z : \tilde{\Phi}$ $\mid \texttt{type}\ t \mid \texttt{type}\ t = \tilde{\tau}$	Module component type

Figure 2: Syntax

$p ::= z_i \mid p.z$	Type access path
$T ::= \tau \mid \forall(\alpha)\ T$	Core type scheme
$\tau ::= t_i \mid p.t \mid \alpha \mid \tau \to \tau \mid \mu(\alpha)\ \tau \mid \langle\, \omega\, \rangle$	Core type
$\omega ::= \epsilon \mid \alpha \mid y : \tau;\ \omega$	Object type
$\Phi ::= \phi \mid \forall(\alpha)\ \Phi$	Module type scheme
$\phi ::= \texttt{functor}\ (z_i : \Phi) \to \Phi \mid \texttt{sig}\ (\tau)\ \chi\ \texttt{end}$	Module type
$\chi ::= \epsilon \mid \psi;\ \chi$	Module body type
$\psi ::= \texttt{val}\ x_i : T \mid \texttt{method}\ y \mid \texttt{module}\ z_i : \Phi$ $\mid \texttt{type}\ t_i \mid \texttt{type}\ t_i = \tau$	Module component type

Figure 3: Types

3.2 Typing rules

More precise types, presented in figure 3, are required for typing: names may be ambiguous and are thus annotated by a stamp, forming identifiers (this is the only difference between the syntactic types previously described and the types in this figure.) Identifiers are subject to α-conversion. However, only the stamp part can be renamed: names must still remain fixed. Types are considered modulo renaming of their bound identifiers.

An environment is a sequence containing signature components and bindings of a type variable to a type. An identifier cannot be bound twice in an environment (*i.e.* a name cannot be bound twice with the same stamp, but it is possible to bind two identifiers with the same name and different stamps.)

$$A ::= \epsilon \mid A;\ \psi \mid A;\ \big(\texttt{var}\ \alpha_i = \tau\big)$$

The notation $A(n)$ stands for the rightmost binding of an identifier of name n in the environment A. We write $\mathrm{FV}(A)$ and $\mathrm{FV}(\tau)$ for the free type variables of respectively an environment A and a type τ. The operation $A + \chi$ discharges the binding of the module body χ inside the environment A. It is defined by $A + \big(\psi;\ \chi\big) = \big(A;\ \psi\big) + \chi$ and $A + \epsilon = A$. The operation $\phi +_A \phi'$ concatenates the module signatures ϕ and ϕ'. It is defined by $\big(\texttt{sig}\ (\tau)\ \chi\ \texttt{end}\big) +_A \big(\texttt{sig}\ (\tau')\ \chi'\ \texttt{end}\big) = \texttt{sig}\ (\tau')\ \chi + \chi'\ \texttt{end}$ when $A \vdash \tau \approx \tau'$, where the operation $\chi + \chi'$ is defined by $\big(\psi;\ \chi\big) + \chi' = \psi;\ \big(\chi + \chi'\big)$ and $\epsilon + \chi' = \chi'$.

The typing rules assign module types to module expressions ($A \vdash m : \Phi$.) They make use of a notion of type equivalence $A \vdash T \approx T'$ and a subtyping relation between module type schemes $A \vdash \Phi <: \Phi'$. Both judgments are standard (except their extension to type schemes.) They are defined in appendix A. The relation $\approx$ is extended to module types ($A \vdash \Phi \approx \Phi'$) as the equivalence relation associated to the preorder $<:$. The translation of types $A \vdash \tilde{T} : T$ and $A \vdash \tilde{\Phi} : \Phi$ (annotating names with stamps) is presented in appendix B.

So as to make the typing rules more readable, some implicit assumptions are systematically made: all types appearing in the typing rules must be well-formed, and the bound variables of module types must be disjoint from the bound variables of the environment. Well-formedness of types is defined in appendix C: all type constructors appearing free in a type or a module type must be bound in the environment; moreover, two components of the same signature cannot have the same name.

Figure 4 gives the typing rules for the core module language. These typing rules are usual. In rule Variable, the type of the module z is enriched in order to reflect that its type components come from the module identifier z_i, using the strengthening operator Φ/z_i described in appendix D. In rule Projection, the type Φ of module z_i may depend on other components of the structure χ. The type of $\tilde{p}.z$ is thus a type Φ', equivalent to Φ but not depending on these components. Other rules are straightforward.

Figure 5 recalls the typing rules for instantiation and generalization of module types previously presented in section 2. The rules Inst and Gen are the direct transposition to module types of the standard rules for instantiation and generalization of type schemes. The rule Quant prevents the generalization of type variables bound by the syntactic type variable α during the typing of the module expression m.

The typing rules for the class components of structures are contained in figure 6. Rule Inherit is mostly syntactic sugar: rule Then-2 accomplish the actual merge of two module signatures. A method is typed like a function (rule Method); the argument type of this function must be the self type τ, while its result type must be the type of method y in τ.

The typing rule for the operator `new` is adapted from Objective ML. The type of the object `new` m is the self type τ, and all its methods must be defined (rightmost premise.)

(New)

$$\frac{A \vdash m : \big(\texttt{sig}\ (\tau)\ \chi\ \texttt{end}\big) \qquad A \vdash \tau \approx \langle\, y : \tau_y \,\rangle^{(\texttt{method}\ y) \in \chi}}{A \vdash \big(\texttt{new}\ m\big) : \tau}$$

4 Impact of the extension

Our extension is conservative. In particular, it preserves the expressiveness of the extended module system.

The extension allows to make type parameterization of functors implicit.

$$\text{(Variable)}\quad \frac{A(z) = (\texttt{module}\ z_i : \Phi)}{A \vdash z : \Phi/z_i}$$

$$\text{(Projection)}\quad \frac{A \vdash \tilde{p} : (\texttt{sig}\ (\tau)\ \chi\ \texttt{end}) \qquad (\texttt{module}\ z_i : \Phi) \in \chi \qquad A + \chi \vdash \Phi \approx \Phi'}{A \vdash \tilde{p}.z : \Phi'}$$

$$\text{(Structure)}\quad \frac{A \vdash b : \phi}{A \vdash (\texttt{struct}\ b\ \texttt{end}) : \phi}$$

$$\text{(Constraint)}\quad \frac{A \vdash m : \Phi' \qquad A \vdash \tilde{\Phi} : \Phi \qquad A \vdash \Phi' <: \Phi}{A \vdash (m : \tilde{\Phi}) : \Phi}$$

$$\text{(Functor)}\quad \frac{A \vdash \tilde{\Phi}' : \Phi' \qquad A;\ (\texttt{module}\ z_i : \Phi') \vdash m : \Phi}{A \vdash (\texttt{functor}\ (z : \tilde{\Phi}') \to m) : (\texttt{functor}\ (z_i : \Phi') \to \Phi)}$$

$$\text{(App)}\quad \frac{A \vdash m : (\texttt{functor}\ (z_i : \Phi_1') \to \Phi_1) \qquad A \vdash m' : \Phi_2' \qquad A \vdash \Phi_2' <: \Phi_1' \qquad A;\ (\texttt{module}\ z_i : \Phi_2') \vdash \Phi_1 \approx \Phi_2}{A \vdash (m\ m') : \Phi_2}$$

$$\text{(Empty)}\quad \frac{}{A \vdash \epsilon : (\texttt{sig}\ (\tau)\ \epsilon\ \texttt{end})}$$

$$\text{(Then-1)}\quad \frac{A \vdash d : \psi \qquad A;\ \psi \vdash b : \texttt{sig}\ (\tau)\ \chi\ \texttt{end}}{A \vdash (d;\ b) : (\texttt{sig}\ (\tau)\ \psi;\ \chi\ \texttt{end})}$$

$$\text{(Value)}\quad \frac{A \vdash e : T}{A \vdash (\texttt{val}\ x = e) : (\texttt{val}\ x_i : T)}$$

$$\text{(Module)}\quad \frac{A \vdash m : \Phi}{A \vdash (\texttt{module}\ z = m) : (\texttt{module}\ z_i : \Phi)}$$

$$\text{(Type)}\quad \frac{A \vdash \tilde{\tau} : \tau \qquad \mathrm{FV}(\tau) = \emptyset}{A \vdash (\texttt{type}\ t = \tilde{\tau}) : (\texttt{type}\ t_i = \tau)}$$

Figure 4: Typing rules for core structure and module expressions

$$\text{(Inst)}\quad \frac{A \vdash m : \forall(\alpha)\ \Phi}{A \vdash m : \Phi[\tau/\alpha]}$$

$$\text{(Gen)}\quad \frac{A \vdash m : \Phi \qquad \alpha \notin \mathrm{FV}(A)}{A \vdash m : \forall(\alpha)\ \Phi}$$

$$\text{(Quant)}\quad \frac{A;\ (\texttt{var}\ \alpha_i = \tau) \vdash m : \Phi}{A \vdash ([\alpha]\ m) : \Phi}$$

Figure 5: Typing rule for module type quantification

$$\frac{A \vdash d : \phi \qquad A + \phi \vdash b : \phi'}{A \vdash \big(d;\, b\big) : \big(\phi +_A \phi'\big)}\ (\textsc{Then-2}) \qquad \frac{A \vdash m : \phi}{A \vdash \big(\texttt{inherit}\ m\big) : \phi}\ (\textsc{Inherit})$$

$$\frac{A \vdash \big(\texttt{fun}\ x \to e\big) : \tau' \to \tau \qquad A \vdash \tau' \approx \langle\, y : \tau;\, \omega \,\rangle}{A \vdash \big(\texttt{method}\ y = \texttt{fun}\ x \to e\big) : \big(\texttt{sig}\ (\tau')\ \texttt{method}\ y\ \texttt{end}\big)}\ (\textsc{Method})$$

Figure 6: Typing rules for inheritance and methods

Without the extension, abstract types in the signature of functor arguments are used for parameterizing modules over types. This is demonstrated by the following example, the type of a functor implementing sets.

$$\begin{array}{l}\texttt{functor}\ (\mathit{ord} : \texttt{sig type}\ u;\ \texttt{val}\ \mathit{compare} : u \to u \to \mathit{int}\ \texttt{end}) \to \\ \qquad \texttt{sig type}\ t;\ \texttt{val}\ \mathit{empty} : t;\ \texttt{val}\ \mathit{add} : t \to \mathit{ord}.u \to t;\ \ldots\ \texttt{end}\end{array}$$

When applying a functor of this type, both the type u of the set elements and the function *compare* for comparing elements must be provided. With our extension, the type u is not required anymore: the functor can be rewritten to have the following type.

$$\begin{array}{l}\forall(\alpha)\ \big(\texttt{functor}\ (\mathit{ord} : \texttt{sig val}\ \mathit{compare} : \alpha \to \alpha \to \mathit{int}\ \texttt{end}) \to \\ \qquad\qquad \texttt{sig type}\ \alpha\ t;\ \texttt{val}\ \mathit{empty} : \alpha\ t;\ \texttt{val}\ \mathit{add} : \alpha\ t \to \alpha \to \alpha\ t;\ \ldots\ \texttt{end}\big)\end{array}$$

Then, when applying the functor, only the comparison function will have to be provided; the type of the elements will be inferred. This type can even be polymorphic, which is not possible without the extension. Indeed, the result of the application of a functor of the preceding type to a structure of type

$$\big(\texttt{sig val}\ \mathit{compare} : \forall(\alpha)\ \alpha \to \alpha \to \mathit{int}\ \texttt{end}\big)$$

have type

$$\big(\forall(\alpha)\ \texttt{sig type}\ \alpha\ t;\ \texttt{val}\ \mathit{empty} : \alpha\ t;\ \texttt{val}\ \mathit{add} : \alpha\ t \to \alpha \to \alpha\ t;\ \ldots\ \texttt{end}\big)$$

which is equivalent to type

$$\big(\texttt{sig type}\ \alpha\ t;\ \texttt{val}\ \mathit{empty} : \forall(\alpha)\ \alpha\ t;\ \texttt{val}\ \mathit{add} : \forall(\alpha)\ \alpha\ t \to \alpha \to \alpha\ t;\ \ldots\ \texttt{end}\big)$$

In the second functor type, the type t have a type parameter, which is constrained to be the quantification variable α[2]. It is indeed necessary to make explicit that the type t depends on the variable α: let us for instance assume that our sets have a function *max* that returns the greatest element of the set. In the last module type, the type $\forall(\alpha)\ t \to \alpha$ would not be correct for this function, while $\forall(\alpha)\ \alpha\ t \to \alpha$ is.

[2] Constrained type abbreviations are presented in [5]; another variant was previously used in [1].

5 Type inference

This extension does not add much complexity to type inference of modules. So that type constructors do not get unified to type variables bound outside of their scope, unification under a prefix must be used. This is simple, and easy to implement (and is actually implemented in Objective Caml, that allows module components to have non-closed types.) Other changes to type inference should be straightforward.

6 Imperative features

So as to ensure type safety, value-only polymorphism [7] should *a priori* be extended to module expressions. That is, only module expressions that are syntactically module values (paths and functors, but neither structures nor functor applications) should be generalized[3]. The rational is that the evaluation of these expressions cannot produce side-effects. However, this results in a important loss of expressiveness: the example of polymorphic sets in section 4 is not valid anymore. This does not appear to be a problem for classes though, as classes are usually functors.

A way to circumvent this problem is to enrich module types, so as to distinguish functors that evaluates without producing any side-effects from other functors. Indeed, it is sound to generalize the type of the result of an application of such a functor.

7 Limitations and future work

Even though our proposal combines classes and modules in one construction, the choice between the two programming styles remains: one can either provide an abstract type and functions manipulating values of this type:

$\texttt{module}\ \mathit{stack} : \texttt{sig type}\ \alpha\ t; \ldots\ \texttt{end} = \texttt{struct}$
$\quad \texttt{type}\ \alpha\ t = \alpha\ \mathit{list};$
$\quad \texttt{val}\ \mathit{empty} = [\,];$
$\quad \texttt{val}\ \mathit{push}\ l\ x = x :: l;$
$\quad \texttt{val}\ \mathit{pop}\ (x :: l) = (x, l);$
$\quad \ldots$
$\texttt{end}$

[3]The rule for module subtyping corresponding to generalization should also be modified.

or define some methods providing the same functionalities:

```
module stack = [α] struct
   val impl = ([] : α list);
   method push = fun y → fun x → {⟨impl = x :: impl⟩};
   method pop = fun y → let (x :: l) = impl in (x, {⟨impl = l⟩});
   ...
end
```

(The expression $\{\langle x = e\rangle\}$ evaluates to a copy of self where the value component x is bound to the result of the evaluation of expression e.)

In this case, it is actually possible for the same module to provide both styles.

```
module stack = [α'] struct
   type α t = α list;
   val empty = [];
   val push x l = x :: l;
   val pop (x :: l) = (x, l);
   ...;
   val impl = (empty : α' t);
   method push = fun y → fun x → {⟨impl = push impl x⟩};
   method pop = fun y → let (x, l) = pop impl in (x, {⟨impl = l⟩});
   ...
end
```

However, for each function, a corresponding method must still be written, consisting actually (at least in this case) in just stub code calling the function.

It would be interesting to be able to write any module so that it can be used for both styles of programming, without having any stub code to write. As a first step toward this goal, one can notice that it is possible to automatically generate a module providing the functionalities of the first module *stack* from the class *stack*:

```
module stack' : sig type α t; ... end = struct
   type α t = α stack;
   val empty = new stack;
   val push s x = s#push x;
   val pop s = s#pop;
   ...
end
```

Furthermore, this wrapping can easily be generalized to any class definition.

However, classes written as a wrapper around an abstract datatype seems to be a common pattern, which makes the previous wrapping much less interesting. As an example, one can consider the implementation of sets as balanced trees. It seems more natural to define balanced trees using an abstract datatype (so that they can be manipulated using pattern matching), and then eventually provide a class that hides this implementation. But it is not clear how such a class could be generated automatically in the general case. If the module defines several datatypes, which one should be chosen ? What should be done of functions whose first argument does not have the right type ? The translation of functions that takes two arguments of the same type, such as set union, is also complicated, as this means that an object must access the implementation of another object (which should be private.)

8 Related works

Reppy and Riecke already proposed to encode classes in modules [6]. They use Object ML, an extension of SML with simple objects and subtyping, but no classes. Their encoding requires no further extension. As a consequence, it is somewhat verbose. For instance, a creation function must be explicitly defined; this function creates a new object and explicitly fills the different members of the object. Some syntactic sugar should be provided in order to simplify the encoding, but it is not clear how this could be done.

Type quantification of module expressions has also been suggested by Stefan Kahrs [3]. His proposal seems to be equivalent to ours, but differs in several points. Type variables are explicitly introduced. A type definition in the scope of a type variable implicitly depends on this type variable (which can appear on the right-hand side of the type definition.) As a consequence, when used outside of the scope of the type variable, the defined type constructor takes an additional parameter.

9 Conclusion

We have been able to smoothly combine the class language of Objective ML with a module language. This required a small extension to module typing: structure components are allowed to have non-closed types, and module types can be quantified.

While combining classes and modules reduces the complexity of a language providing both, in our proposal modules must still be used either as usual modules or as classes. It is still not possible to use the same definition both as a module or a class.

References

[1] 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.

[2] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st symposium Principles of Programming Languages*, pages 123–137. ACM Press, 1994.

[3] Stefan Kahrs. First-class polymorphism for ML. Technical report, University of Edinburgh, 1993.

[4] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[5] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. To appear in *Theory and Practice of Object Systems*, 1997.

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

[7] Andrew K. Wright. Polymorphism for imperative languages without imperative types. Technical report TR93-200, Rice University, 1993.

A Equivalence and subtyping relations

Type access path

$$\frac{\big(\mathtt{module}\ z_i : \Phi\big) \in A}{A \vdash z_i : \Phi} \qquad \frac{A \vdash p : \big(\mathtt{sig}\ (\tau)\ \chi\ \mathtt{end}\big) \qquad \big(\mathtt{module}\ z_i : \Phi\big) \in \chi \qquad A + \chi \vdash \Phi \approx \Phi'}{A \vdash p.z : \Phi'}$$

$$\frac{A \vdash p : \forall(\alpha)\ \Phi}{A \vdash p : \Phi[\tau/\alpha]}$$

Type equivalence (rules for congruence, reflexivity, symmetry, transitivity omitted)

$$\frac{\big(\mathtt{type}\ t_i = \tau\big) \in A}{A \vdash t_i \approx \tau} \qquad \frac{A \vdash p : \mathtt{sig}\ (\tau)\ \chi\ \mathtt{end} \qquad \big(\mathtt{type}\ t_i = \tau\big) \in \chi}{A \vdash p.t \approx \tau}$$

Module subtyping

$$\frac{A \vdash \tau_1 \approx \tau_2}{A \vdash \tau_1 <: \tau_2} \qquad \frac{A \vdash \Phi_2' <: \Phi_1' \qquad A;\ \big(\mathtt{module}\ z_i : \Phi_2'\big) \vdash \Phi_1 <: \Phi_2}{A \vdash \mathtt{functor}\ (z_i : \Phi_1') \to \Phi_1 <: \mathtt{functor}\ (z_i : \Phi_2') \to \Phi_2}$$

$$\frac{A \vdash \tau_1 \approx \tau_2 \qquad \forall d_2 \in \chi_2 \cdot \exists d_1 \in \chi_1 \cdot A + \chi_1 \vdash d_1 <: d_2}{A \vdash (\mathtt{sig}\ (\tau_1)\ \chi_1\ \mathtt{end}) <: (\mathtt{sig}\ (\tau_2)\ \chi_2\ \mathtt{end})}$$

$$\frac{A \vdash T_1 <: T_2}{A \vdash (\mathtt{val}\ x_i : T_1) <: (\mathtt{val}\ x_i : T_2)} \qquad \frac{A \vdash \Phi_1 <: \Phi_2}{A \vdash (\mathtt{module}\ z_i : \Phi_1) <: (\mathtt{module}\ z_i : \Phi_2)}$$

$$\frac{}{A \vdash (\mathtt{method}\ y) <: (\mathtt{method}\ y)}$$

$$\frac{}{A \vdash (\mathtt{type}\ t_i) <: (\mathtt{type}\ t_i)} \qquad \frac{}{A \vdash (\mathtt{type}\ t_i = \tau) <: (\mathtt{type}\ t_i)}$$

$$\frac{A \vdash \tau \approx \tau'}{A \vdash (\mathtt{type}\ t_i = \tau) <: (\mathtt{type}\ t_i = \tau')} \qquad \frac{A \vdash t_i \approx \tau}{A \vdash (\mathtt{type}\ t_i) <: (\mathtt{type}\ t_i = \tau)}$$

$$\frac{A \vdash \Phi_1[\tau/\alpha] <: \Phi_2}{A \vdash \forall(\alpha)\ \Phi_1 <: \Phi_2} \qquad \frac{\begin{array}{c} A \vdash \Phi_1 <: \Phi_2 \\ \alpha \notin \mathrm{FV}(A) \end{array}}{A \vdash \Phi_1 <: \forall(\alpha)\ \Phi_2}$$

$$\frac{A \vdash T_1[\tau/\alpha] <: T_2}{A \vdash \forall(\alpha)\ T_1 <: T_2} \qquad \frac{\begin{array}{c} A \vdash T_1 <: T_2 \\ \alpha \notin \mathrm{FV}(A) \end{array}}{A \vdash T_1 <: \forall(\alpha)\ T_2}$$

Module type scheme equivalence

$$\frac{A \vdash \Phi_1 <: \Phi_2 \qquad A \vdash \Phi_2 <: \Phi_1}{A \vdash \Phi_1 \approx \Phi_2}$$

B Type translation

Core type (structural rules omitted)

(TYPE-NAME)
$$\frac{\big(\mathtt{type}\ t_i[= \tau\ ]\big) \in A}{A \vdash t : t_i}$$

(TYPE-PROJECTION)
$$\frac{A \vdash \tilde{p} : \mathtt{sig}\ (\tau)\ \chi\ \mathtt{end} \qquad \big(\mathtt{type}\ t_i = \tau\big) \in \chi}{A \vdash \tilde{p}.t : \tau}$$

(TYPE-VAR)
$$\frac{A(\alpha) = (\mathtt{var}\ \alpha_i = \tau)}{A \vdash \alpha : \tau}$$

(TYPE-QUANT)
$$\frac{A;\ \big(\mathtt{var}\ \alpha_i = \alpha'\big) \vdash \tilde{T} : T \qquad \alpha' \notin \mathrm{FV}(A)}{A \vdash \forall(\alpha)\ \tilde{T} : \forall(\alpha')\ T}$$

Signature

(SIG-EMPTY)
$$\frac{}{A \vdash \epsilon : \epsilon}$$

(SIG-THEN)
$$\frac{A \vdash \tilde{\psi} : \psi \qquad A;\ \psi \vdash \tilde{\chi} : \chi}{A \vdash \big(\tilde{\psi};\ \tilde{\chi}\big) : \big(\psi;\ \chi\big)}$$

(SIG-VALUE)
$$\frac{A \vdash \tilde{T} : T}{A \vdash \big(\mathtt{val}\ x : \tilde{T}\big) : \big(\mathtt{val}\ x_i = T\big)}$$

(SIG-MODULE)
$$\frac{A \vdash \tilde{\Phi} : \Phi}{A \vdash \big(\mathtt{module}\ z : \tilde{\Phi}\big) : \big(\mathtt{module}\ z_i : \Phi\big)}$$

(SIG-MANIFEST-TYPE)
$$\frac{A \vdash \tilde{\tau} : \tau \qquad \mathrm{FV}(\tau) = \emptyset}{A \vdash \big(\mathtt{type}\ t = \tilde{\tau}\big) : \big(\mathtt{type}\ t_i = \tau\big)}$$

$$\frac{}{A \vdash (\texttt{type}\ t) : (\texttt{type}\ t_i)}\ (\textsc{Sig-Abstract-Type}) \qquad \frac{}{A \vdash (\texttt{method}\ y) : (\texttt{method}\ y)}\ (\textsc{Sig-Method})$$

$$\frac{A \vdash \tau : \tau' \qquad A \vdash \chi : \chi'}{A \vdash (\texttt{sig}\ (\tau)\ \chi\ \texttt{end}) : (\texttt{sig}\ (\tau')\ \chi'\ \texttt{end})}\ (\textsc{Signature})$$

$$\frac{A \vdash \tilde{\Phi}' : \Phi' \qquad A;\ (\texttt{module}\ z_i : \Phi') \vdash \tilde{\Phi} : \Phi}{A \vdash (\texttt{functor}\ (z : \tilde{\Phi}') \to \tilde{\Phi}) : (\texttt{functor}\ (z : \Phi') \to \Phi)}\ (\textsc{Sig-Functor})$$

$$\frac{A;\ (\texttt{var}\ \alpha_i = \alpha') \vdash \tilde{\Phi} : \Phi \qquad \alpha' \notin \mathrm{FV}(A)}{A \vdash \forall(\alpha)\ \tilde{\Phi} : \forall(\alpha')\ \Phi}\ (\textsc{Sig-Quant})$$

C Type well-formedness

Type well-formedness $(A \vdash T \text{ type})$

(structural rules omitted)

$$\frac{(\texttt{type}\ t_i[=\tau\,]) \in A}{A \vdash t_i \text{ type}} \qquad \frac{A \vdash p : \texttt{sig}\ (\tau)\ \chi\ \texttt{end} \qquad (\texttt{type}\ t_i = \tau) \in \chi}{A \vdash p.t \text{ type}}$$

Module type well-formedness $(A \vdash \Phi \text{ module type})$

$$\frac{}{A \vdash \epsilon \text{ decl}} \qquad \frac{A \vdash T \text{ type} \qquad A \vdash \chi \text{ decl}}{A \vdash (\texttt{val}\ x_i = T;\ \chi) \text{ decl}} \qquad \frac{A \vdash \chi \text{ decl}}{A \vdash (\texttt{method}\ y;\ \chi) \text{ decl}}$$

$$\frac{A \vdash \tau \text{ type} \qquad A;\ (\texttt{type}\ t_i = \tau) \vdash \chi \text{ decl}}{A \vdash (\texttt{type}\ t_i = \tau;\ \chi) \text{ decl}} \qquad \frac{A;\ (\texttt{type}\ t_i) \vdash \chi \text{ decl}}{A \vdash (\texttt{type}\ t_i;\ \chi) \text{ decl}}$$

$$\frac{A \vdash \phi \text{ module type} \qquad A;\ (\texttt{module}\ z_i : \phi) \vdash \chi \text{ decl}}{A \vdash (\texttt{module}\ z_i : \phi;\ \chi) \text{ decl}}$$

$$\frac{\begin{array}{c} A \vdash \tau \text{ type} \qquad A \vdash \chi \text{ decl} \\ \text{no name bound twice in } \chi \end{array}}{A \vdash (\texttt{sig}\ (\tau)\ \chi\ \texttt{end}) \text{ module type}} \qquad \frac{\begin{array}{c} A \vdash \Phi' \text{ module type} \\ A;\ (\texttt{module}\ z_i : \Phi') \vdash \Phi \text{ module type} \end{array}}{A \vdash (\texttt{functor}\ (z_i : \Phi') \to \Phi) \text{ module type}}$$

$$\frac{A \vdash \Phi \text{ module type} \qquad \alpha \notin \mathrm{FV}(A)}{A \vdash \forall(\alpha)\ \Phi \text{ module type}}$$

D Strengthening

(structural rules omitted)

$$
\begin{array}{rcl}
(\texttt{functor}\ (z_i : \Phi') \to \Phi)/p & \longrightarrow & \texttt{functor}\ (z_i : \Phi') \to \Phi/(p\ z_i) \\
(\texttt{sig}\ (\tau)\ \chi\ \texttt{end})/p & \longrightarrow & \texttt{sig}\ (\tau)\ \chi/p\ \texttt{end} \\
(\texttt{module}\ z_i : \Phi)/p & \longrightarrow & \texttt{module}\ z_i : \Phi/p.z \\
(\texttt{type}\ t_i)/p & \longrightarrow & \texttt{type}\ t_i = p.t \\
(\texttt{type}\ t_i = \tau')/p & \longrightarrow & \texttt{type}\ t_i = p.t
\end{array}
$$