Mixing the Objective Caml and C# Programming Models in the .NET Framework

Emmanuel Chailloux[1], Grégoire Henry[1], and Raphaël Montelatici[2]

[1] Equipe Preuves, Programmes et Systèmes (CNRS UMR 7126)
Université Pierre et Marie Curie (Paris 6) - 4 place Jussieu, 75005 Paris, France
Emmanuel.Chailloux@pps.jussieu.fr,Gregoire.Henry@pps.jussieu.fr
[2] Equipe Preuves, Programmes et Systèmes (CNRS UMR 7126)
Université Denis Diderot (Paris 7) - 2 place Jussieu, 75005 Paris, France
Raphael.Montelatici@pps.jussieu.fr

Abstract. We present a new code generator, called `O'Jacaré.net`, to inter-operate between C# and Objective Caml through their object models. `O'Jacaré.net` defines a basic IDL *(Interface Definition Language)* that describes classes and interfaces in order to communicate between Objective Caml and C#. `O'Jacaré.net` generates all needed wrapper classes and takes advantage of static type checking in both worlds. Although the IDL intersects these two object models, `O'Jacaré.net` allows to combine features from both.

1 Introduction

The .NET platform claims to be a melting pot that allows the integration of different languages in a common framework, sharing a common type system, CTS, and a runtime environment, CLR (*Common Language Runtime*). Each .NET compiler generates portable MSIL byte-code (*MicroSoft Intermediate Language*). By assuming compliance to the CTS type system, components inter-operate safely.

The .NET framework is actually well suited for object-oriented languages which have an object model close to the one of C# or Java. Unfortunately, languages with other kinds of object models, type systems or supporting different programming paradigms (such as functional programming . . .) do not fit in .NET as well as C# does. Writing .NET compilers for them requires much more efforts.

However, the .NET framework still gives us a good opportunity to experiment inter-operability between two languages as different as Objective Caml[1] (shortened as O'Caml) and C#. O'Caml is an ML dialect: it is a functional/imperative statically typed language, featuring parametric polymorphism, an exception mechanism, an object layer and parameterized modules. By achieving inter-operability, each language gains access to a wider set of libraries and programmers take advantage of a richer programming model.

We use the experimental OCamIL compiler[2], which compiles the whole O'Caml distribution (including toplevel) to .NET managed code. We intend to communicate between O'Caml and C# by means of their respective object

models. Difficulties arise because neither the type system nor the object model of O'Caml natively fit in the CTS. O'Caml objects cannot be directly compiled to CTS objects. Communication cannot be direct: C# and O'Caml objects have to be interfaced. We use an IDL (Interface Description Language) and a code generator called `O'Jacaré.net`. It is based on our previous work[3] on interoperability of O'Caml and Java.

We first describe the O'Caml object model and compare it to the C# model. We then introduce `O'Jacaré.net`, using a small example as an illustration of its features. The last section is dedicated to expressiveness issues. We show that the combination of two object models allows to take advantage of features of both, and also discuss the current limitations of `O'Jacaré.net`, giving hints on how they can be solved in further developments.

2 Comparing Object Models

O'Caml is a statically typed language based on a functional and imperative kernel. It also integrates a class-based object-oriented extension in its type system, for which inheritance relation and subtyping relation for classes are well distinguished[4]. One key feature of O'Caml type system is type inference. The programmer does not annotate programs with typing indications: the compiler gives each expression the most general type it cans.

A class declaration defines:

- a new type abbreviation of an object type,
- a constructor function to build class instances.

An object type is characterized by the name and the type of its methods. For instance, the following type can be inferred for class instances which declare `moveto` and `toString` methods:

```
< moveto : (int * int) -> unit; toString : unit -> string >
```

At each method call site, static typing checks that the type of the receiving instance is an object type and that it contains the relevant method name with a compatible type. The following example is correct if the class `point` defines (or inherits) a method `moveto` expecting a couple of integers as argument. Within the O'Caml type inference, the most general types given to objects are expressed by means of "open" types (`<..>`). The function `f` can be used with any object having a method `moveto` (`'a` denotes a universally quantified type variable):

method call	functional-object style
let p = **new** point(1,1);; p#moveto(10,2);;	**#** **let** f o = o **#** moveto (10,20);; **val** f : < moveto : int * int –> 'a; .. > –> 'a

Here are some of O'Caml object model most important characteristics:

- Class declarations allow multiple inheritance and parametric classes.
- Method overloading is not supported.
- The methods binding is always delayed.

The C# language model is well known and will not be described here. We compare its main features with O'Caml in the following table:

Features	C#	O'Caml	Features	C#	O'Caml
classes	√	√	inheritance ≡ sub-typing?	yes	no
late binding	√	√	overloading	√	3
early binding	√	1	multiple inheritance	4	√
static typing	√	√	parametric classes	5	√
dynamic typing	√	2	packages/modules	6	6
sub-typing	√	√			

1) static methods are global functions of a O'Caml module and class variables are global declarations;

2) no downcast in O'Caml (only available in the coca-ml [5] extension);

3) no overloading in O'Caml but the type of `self` can appear in the type of a method eventually overridden in a subclass;

4) no multiple inheritance for C# classes, only for interfaces;

5) generics[6] are expected in C# 2.0;

6) simple modules of O'Caml correspond to public parts of C# namespaces; there is no parameterized modules in C#.

The intersection of these two models corresponds to a basic class-based language, where method calls are delayed, and inheritance and subtyping relations are equivalent. Concerning type system, there is no overloading and no binary methods. For the sake of simplicity, there is no multiple inheritance nor parametric classes. This model inspires a basic IDL for interfacing C# and O'Caml classes.

3 Introducing `O'Jacaré.net`

`O'Jacaré.net` is based on our previous work `O'Jacaré` on O'Caml and Java. Its purpose was to use Java objects in O'Caml. We encountered difficulties with the management of two different runtimes (Java runtime and O'Caml runtime), especially for handling threads and garbage collection. Adapting this work to C# and the OCamIL implementation of O'Caml on .NET makes things easier, mainly because there is only one runtime.

`O'Jacaré.net` allows to use C# objects in O'Caml, and O'Caml objects in C# as well.

3.1 C# in O'Caml

Our current communication model between O'Caml and C# affords two levels of communication: as the first level provides a basic encapsulation mechanism of C# objects inside O'Caml objects, the second level adds a callback mechanism that allows to override C# methods in O'Caml using late binding.

Basic encapsulation Starting from the description of classes and interfaces in an IDL file, `O'Jacaré.net` generates wrappers in the target language (here, O'Caml), allowing to allocate objects and call methods upon classes of the foreign language (here, C#) as if those classes where native.

Let us illustrate this mechanism on a small example: we want to handle two C# classes, `Point` and `ColoredPoint`. They are described in the IDL file below.

File `p.idl`

```
package [assembly point] mypack;

class Point {
  int x; int y;

  [name default_point] <init> ();
  [name point] <init> (int,int);
  void moveTo(int,int);
  string toString();
  void display();
  boolean equals(Point);
}
```

```
interface Colored {
  string getColor();
  void setColor(string);
}

class ColoredPoint extends Point
                        implements Colored {
  [name default_colored_point] <init> ();
  [name colored_point] <init> (int,int,string);

  [name equals_pc] boolean equals(ColoredPoint)
}
```

The IDL syntax borrows from Java syntax and is extended with attributes (i.e. for name-aliasing because overloading is not allowed).

For the `p.idl` file, `O'Jacaré.net` generates an O'Caml module, named `p.ml`, that contains:

- 3 class types: `csPoint`, `csColored` and `csColoredPoint` ;
- 3 wrapper classes exposed with previous class types ;
- 4 constructors that allocate and initialize C# objects, wrapping them inside the previous classes.

An example of use is illustrated in an O'Caml toplevel session below (the `equals` method compares two objects using their instance variables `x` and `y`).

O'Caml toplevel session

```
# open P;;
# let p = new point 1 2;;
val p : point = <obj>
# let p2 = new default_point ();;
val p2 : default_point = <obj>
# let pc = new colored_point 3 4 "blue";;
val pc : colored_point = <obj>
# let pc2 = new default_colored_point ();;
val pc2 : default_colored_point = <obj>
# p#toString ();;
- : string = "(1,2)"
```

```
# pc#toString ();;
- : string = "(3,4):blue"
# p#equals (pc :> csPoint);;
- : bool = false
# pc#moveTo 1 2;;
- : unit = ()
# pc#equals p;;
- : bool = true
# pc#equals_pc pc2;;
- : bool = false
```

The type coercion operator `:>` allows to consider the type of an object as a supertype, according to the subtyping relation.

Callback mechanism We go on with the previous example. The C# implementation of the `toString` method of class `ColoredPoint` concatenates the results of a call to the superclass `toString` method and a call to the `getColor` method on itself. We want to redefine the `getColor` method in O'Caml, and so specialize the `toString` method through late binding.

With basic encapsulation, a C# instance of `ColoredPoint` has no knowledge of the O'Caml instance. We need a second level of communication, introduced by the `callback` attribute :

[callback] class ColoredPoint **extends** Point **implements** Colored { ... }

With this attribute, the compilation of the file `p.idl` generates a new file called `ColoredPointStub.cs` and add stub classes to the generated O'Caml file. As shown in right column of the below example, inheriting the stub in O'Caml allows the expected behavior, whereas inheriting the wrapper (left column) does not!

```
# class wrong_ml_colored_point x y c =
  object
    inherit
        colored_point x y c as super
    method getColor () =
        "ML" ^ super#getColor ()
  end;;
class wrong_ml_colored_point :
            int -> int -> string -> csColoredPoint
# let wml_cp =
    new wrong_ml_colored_point 6 7 "green";;
val wml_cp : wrong_ml_colored_point = <obj>
# wml_cp#toString ();;
- : string = "(6,7):green"
```

```
# class ml_colored_point x y c =
  object
    inherit
        callback_colored_point x y c as super
    method getColor () =
        "ML" ^ super#getColor ()
  end;;
class ml_colored_point :
            int -> int -> string -> csColoredPoint
# let ml_cp =
    new ml_colored_point 8 9 "red";;
val ml_cp : ml_colored_point = <obj>
# ml_cp#toString ();;
- : string = "(8,9):MLred"
```

How is this achieved ? The two stubs in C# and O'Caml own a reference upon each other. The C# stub overrides each method as a callback to O'Caml, and the O'Caml stub define each method as a non-virtual call to `ColoredPoint`, the base-class of `ColoredPointStub`. See figure 1 for the complete class diagram.

3.2 Safety considerations

O'Caml ensures execution safety by static typing, but what happens when it uses foreign pieces of code ? We distinguish two kind of errors:

- Runtime errors during C# code execution (dynamic cast errors for instance), that are not a consequence of inter-operation.
- Inconsistency between the IDL and the implementation. For example, C# components described in the IDL may not be available at runtime, or incorrectly described.

The first class of errors will fortunately raise runtime exceptions. They can be considered as "normal" runtime errors. They can be caught by the C# component or, by default, by O'Caml code itself. As O'Caml exceptions and C#

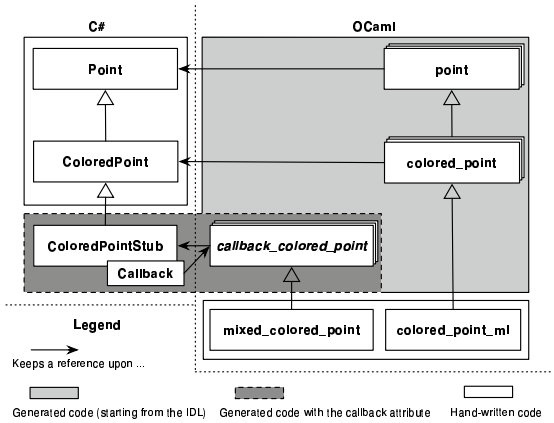


Fig. 1. Relationship between classes

exceptions are both compiled to exceptions of the underlying runtime, they can easily cross languages boundaries.

The second class of errors is a consequence of inter-operation itself. We choose to detect those errors very soon. O'Caml programs that incorrectly use foreign components are detected by static typing at compile time; type checking is done with the assumption that IDL types as correct. This hypothesis can only be checked at runtime with the reflection mechanism. The code which is generated by `O'Jacaré.net` performs tests at startup time, immediately acknowledging the programmer of mismatches between components and IDL files.

3.3 A few words about O'Caml in C#

In order to call O'Caml methods from C# we reuse the technology behind callbacks from C# to O'Caml (see subsection 3.1). Basic encapsulation can similarly be extended with a callback mechanism.

Let us stress the lack of symmetry between O'Caml and C# from an implementation point of view. Whereas C# objects are directly compiled to CTS objects, O'Caml objects are not: the current implementation provides its own mechanisms of inheritance and late binding. Calls from O'Caml to C# directly use reflection mechanisms provided by the .NET runtime but calls from C# to O'Caml have to deal with the peculiarities of O'Caml implementation. A future release of OCamIL may solve this problem.

As O'Caml does not offer any introspection mechanism for type information, we need to adapt our dynamic checking. Even with basic encapsulation, we generate some O'Caml code, that will statically check IDL against O'Caml implementation, and dynamically check it against the generated C# at startup time.

4 Expressiveness and limitations

The introductory example of section 3 only involved a simple form of communication. In this section we try to go a little bit further. We first give a positive result about the expressiveness of the blending of two different object models. Then we apply `O'Jacaré.net` technology to a real example that involves complex communication. It is used as the starting point of a discussion about the actual limitations of `O'Jacaré.net`.

4.1 Combining the two Objects Models

`O'Jacaré.net` allows to partially handle both object models. We illustrate these new possibilities by showing a case of multiple inheritance in O'Caml of C# classes and an example of dynamic type checking (*downcast*) in O'Caml.

Multiple inheritance of C# classes The following example is taken from [7]. We define two class hierarchies in C#: graphical objects and geometrical objects. Each class hierarchy has a class `Rectangle`. The following O'Caml program defines a class inheriting both C# classes.

The file `rect.idl`	The O'Caml program

package mypack;
 	
class Point {
 [**name** point] <**init**> (**int**, **int**);
}
class GraphRectangle {
 [**name** graph_rect] <**init**>(Point, Point);
 string toString();
}
class GeomRectangle {
 [**name** geom_rect] <**init**>(Point, Point);
 double compute_area();
}
	open Rect;;
	
	class geom_graph_rect p1 p2 =
	object
	 inherit geom_rect p1 p2 **as** super_geo
	 inherit graph_rect p1 p2 **as** super_graph
	end;;
	
	let p1 = **new** point 10 10;;
	let p2 = **new** point 20 20;;
	let ggr = **new** geom_graph_rect p1 p2;;
	Printf.printf "area=%g\n" (ggr#compute_area ());;
	Printf.printf "toString=%s\n" (ggr#toString ());;

Downcasting C# objects in O'Caml. O'Caml does not allow any dynamic typing operations on objects, however inter-operating with C# makes them necessary, at least for objects coming from a computation on C# side. The example below builds a list `l` of `csPoint` objects, even though these actually are colored points. For each C# class hierarchy described in an IDL file, `O'Jacaré.net` generates a O'Caml class hierarchy, which root class is denoted by `top`. `O'Jacaré.net` also generates type coercion functions from `top` to the O'Caml type of a C# class. These functions raise an exception in case of type inadequacy.

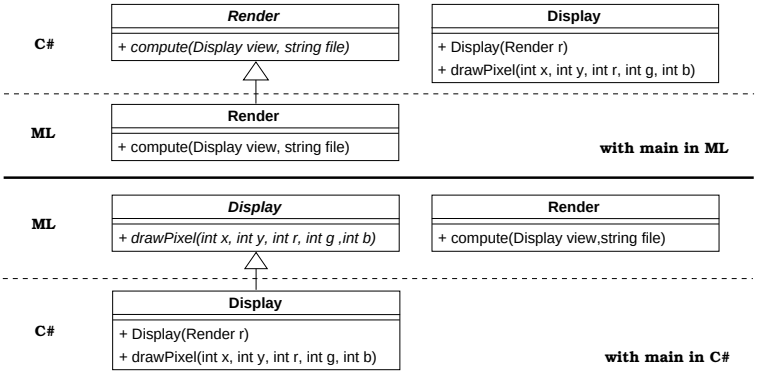
```
let l = [(ml_cp :> csPoint); (wml_cp :> csPoint)];;
val l : csPoint list = <obj>
let lc = List.map (fun x -> csColoredPoint_of_top (x :> top)) l;;
val l : csColoredPoint list = <obj>
```

4.2 Application: a Ray-tracer Program

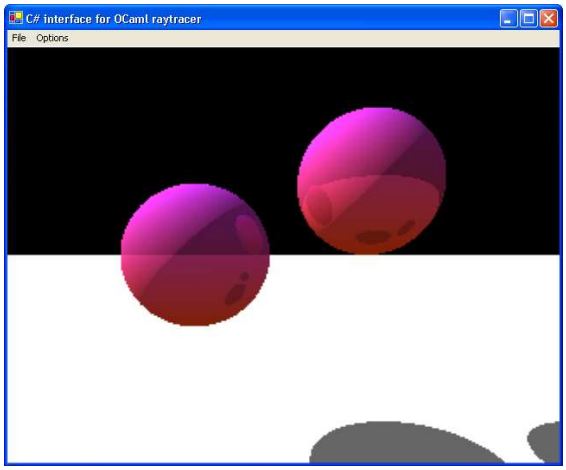
We illustrate inter-operability on the following example: extending an O'Caml program with a graphical interface written in C#. We use the winning entry of the ICFP'2000 programming contest[8] which implements a ray-tracer in O'Caml. Let us state the problem:

- The O'Caml class `Render` defines a `compute` method. This method expects a string (the name of a file that represents the 3D scene to draw) and a class `Display` to render pixels on (thanks to calls to a so-called `drawPixel` method).
- The graphical interface is a class `Display` inheriting from (or holding a reference to an instance of) the root widget `System.Windows.Forms.Form` of .NET windowing API, with a `drawPixel` method. A file dialog helps selecting a 3D scene.

Communication is round tripping between the two components. This can be implemented with `O'Jacaré.net` using cross-language late binding. Two solutions work:



1. The C# interface is parameterized on an abstract class `Render` which defines a `compute` method. The interface constructor expects an instance of this class. When the user chooses a 3D scene file, `compute` is called. The O'Caml program implements the class `Render`. The `Main` method is on O'Caml side: it passes an instance of `Render` to the constructor of `Display` (starting the graphical interface). When called, the `compute` method calls `drawPixel` for each computed pixel.
2. The `Main` method is on C# side: it builds an instance of `Display` which specializes an abstract class defined in O'Caml. Here, because multiple inheritance is not allowed in C#, `Display` only holds a reference to an object that inherits `System.Windows.Forms.Form`. When a scene file is selected, it builds a `Render` object and calls `compute` with the filename and `self`.



4.3 How perfect is the blending ?

`O'Jacaré.net` allows to use components from one language to another, in both ways. However we still cannot claim that this can reduce the two worlds into a single one.

Let us go on with the ray-tracer program. If we had a single world, we could use IDL files to declare the `drawPixel` method from the C# class `Display` and the `compute` method from the O'Caml class `Render`, making them accessible to both components without using the "trick" of redefinition of abstract classes. Unfortunately, IDL files can only be used to expose classes from one language to the other one.

We cannot simulate one world with two IDL files, one for describing the C# `Display` class, then one for the O'Caml `Render` class because those classes are mutually recursive. The point is: the `compute` method expects an instance of `Display`, so the latter IDL needs to describe the O'Caml wrapper for the `Display` generated from the first IDL. This leads to typing errors at C# compile time, because there is no inheritance relationship between the original `Display` and the twice encapsulated `Display`.

5 Conclusion and further work

Our approach differs from MLj[9], SML.NET[10] and F#[11] projects which embed Java or C# object models inside ML dialects. We do not modify O'Caml at all, keeping the specificities of its object model. This leads to a richer model that combines O'Caml polymorphisms with C# dynamic typing.

Each community can use `O'Jacaré.net` to import components from the other one. However `O'Jacaré.net` needs to be improved. Some interesting features from the .NET runtime (such as methods delegates, genericity, ...) should be addressed and made available in the IDL. By making the IDL closer to the CTS, one can also imagine to solve the problem discussed in subsection 4.3.

References

1. Leroy, X.: The Objective Caml system release 3.06 : Documentation and user's manual. Technical report, Inria (2002) on-line version : `http://caml.inria.fr`.
2. Montelatici, R., Chailloux, E., Pagano, B.: Objective Caml on .NET: the OCamIL compiler and toplevel. Technical Report 29, PPS (2004) on-line version : `http://www.pps.jussieu.fr/~montela`.
3. Chailloux, E., Henry, G.: O'Jacaré : une interface objet entre Objective Caml et Java. In: Langages et Modèles à objets (LMO). (2004) `http://www.pps.jussieu.fr/~henry/ojacare`.
4. Remy, D., Vouillon, J.: Objective ML: An effective object-oriented extension to ML. Theory and Practice of Object Systems **4** (1998) 27–50
5. Chailloux, E.: Dynamic object typing in Objective Caml. In: International Lisp Conference. (2002)
6. Kennedy, A., Syme, D.: Design and implementation of generics for the .NET common language runtime. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. (2001) 1–12
7. Chailloux, E., Manoury, P., Pagano, B.: Développement d'Applications avec Objective Caml. 1st edn. O'Reilly (2000) on-line english version : `http://caml.inria.fr`.
8. TeamPLClub: Winning entry of ICFP programming contest (2000) Web page : `http://www.cis.upenn.edu/~sumii/icfp`.
9. Benton, N., Kennedy, A., Russel, G.: Compiling Standard ML to Java Bytecodes. In: Proceedings of the 3rd ACM SIGPLAN Conference on Functional Programming. (1998)
10. Benton, N., Kennedy, A., Russo, C., Russell, G.: sml.net : Functional programming on the .NET CLR (2003) `http://www.cl.cam.ac.uk/Research/TSG/SMLNET/`.
11. Syme, D.: ILX: Extending the .NET common IL for functional language interoperability. Electronic Notes in Theoretical Computer Science **59** (2001) `http://research.microsoft.com/projects/ilx/fsharp.aspx`.