O'JACARÉ.NET

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

Emmanuel Chailloux, Grégoire Henry and Raphaël Montelatici

Equipe PPS (UMR 7126)

1) Université Pierre et Marie Curie (Paris VI)

2) Université Denis Diderot (Paris VII)

http://www.pps.jussieu.fr

MPOOL 2004 - Oslo

Summary

- Motivations
- Objective Caml
- Comparing C# and O'Caml
- Reflection API / external mechanism
- O'Jacaré.Net description:
  - IDL + code generator
  - Usage from O'Caml
  - Usage from C#
- Example : a raytracer.
- Discussion

Motivations

1. Enrich both languages: C# and O'Caml,
   - Make the use of new libraries easy,
2. Preserve safety: O'Caml static typing, GC,
3. Keep the original languages unchanged.

O’Caml features

- Functional language, exceptions, imperative extension,

O’Caml features

- Functional language, exceptions, imperative extension,
- High-level data types + pattern matching,

O'Caml features

- Functional language, exceptions, imperative extension,
- High-level data types + pattern matching,
- Polymorphism + *implicit* typing:
  - statically type-checked,
  - type inference,
  - polymorphic type (most general type is inferred).

O'Caml features

- Functional language, exceptions, imperative extension,
- High-level data types + pattern matching,
- Polymorphism + *implicit* typing:
  - statically type-checked,
  - type inference,
  - polymorphic type (most general type is inferred).
- Multi-paradigm (inside the same typing mechanism):
  - Object-oriented (class structuration),
  - SML-like parametric module,
  - labels and polymorphic variants.

Examples of type inference

- functional type :

```
 let compose f g = fun x -> f (g x);;
```

$(\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta$

- functional type over list :

`List.map` : $(\alpha \to \beta) \to \alpha$ `list` $\to \beta$ `list`

- object and functional type :

```
let toStringNL o = o#toString() ^"\n";;
```

`< toString : unit` $\to$ `string ; .. >` $\to$ `string`

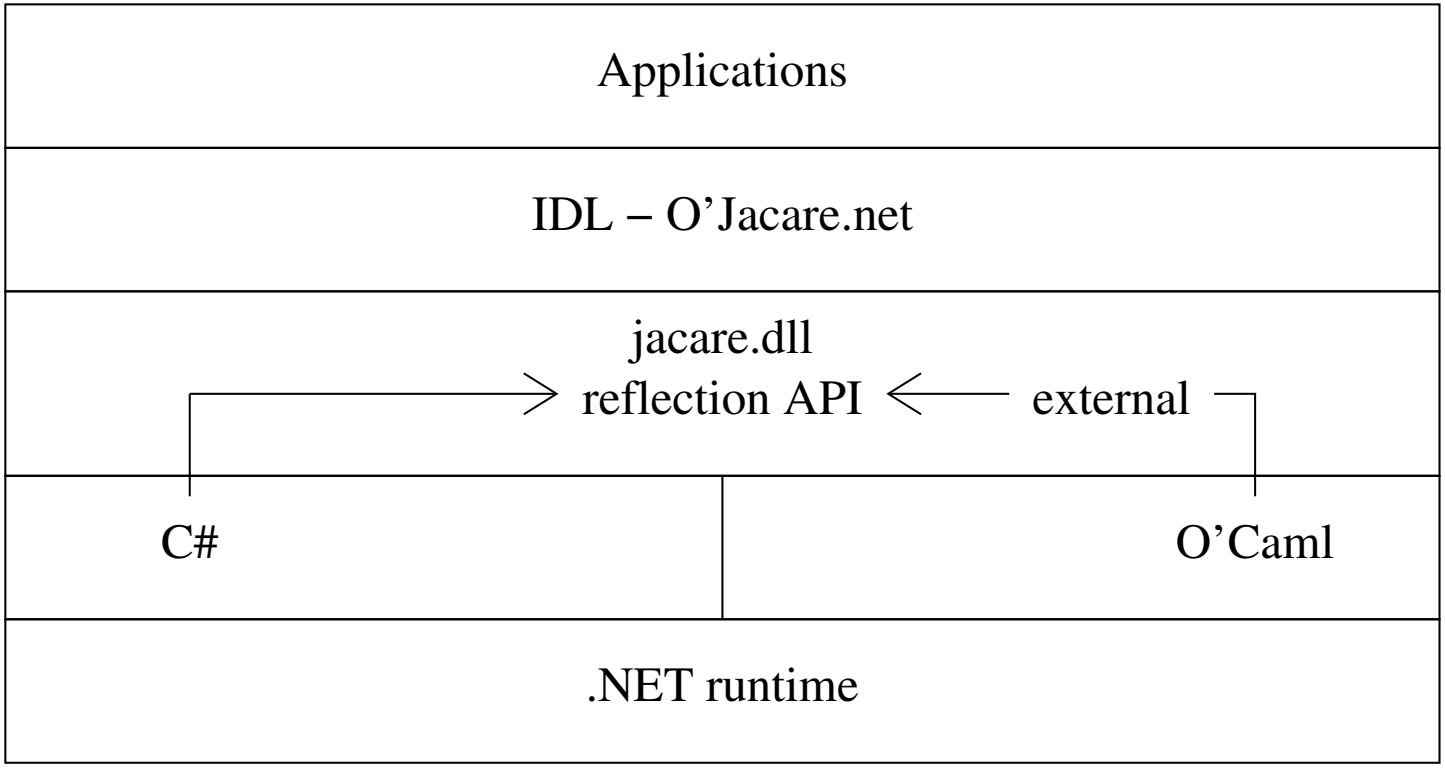
Comparing Object Models

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

O'Caml is not an object language, but has an object-oriented extension

- a class declaration defines a new object type and a constructor function
- object type = method's names and types

Architecture of the interface



Low-level – jacare.dll

C# – Reflection API

- Class searching by name and assembly (System.Type),
- Method identification by name and type of arguments,
- Method calls with an array of arguments.

O'Caml specificities

- O'Caml object are not map compiled to obvious CTS objects,
- Method identification by name only,
- No type introspection on O'Caml side.

Exception

O'Jacaré.Net, a simple IDL - 1/2

Associate one C# object with one O'Caml object

At the intersection of the two models

- Class, abstract class and interface definition,
- Single inheritance for classes,
- Multiple inheritance for interfaces,
- No overloading
  (but an name aliases mechanism),
- No parametric class.

O'Jacaré.Net, a code generator - 2/2

Arguments passing

- by reference for objects (ex : System.Object)
- by copy for base types (ex : int, string)

Typing consistency of the IDL type is checked:

- at compile time against O'Caml
  and against C# if available
- at "load" time between the two implementation
  (introspection)

An IDL file is a simple description of CLR classes.

O'Jacaré.Net : Class Point

File `point.idl`

```
class Point {
    int x;

    [name default_point] <init> ();
    [name point] <init> (int);

    void moveTo(int);
    string toString();
    boolean eq(Point);
}
```

Generates : `point.ml`

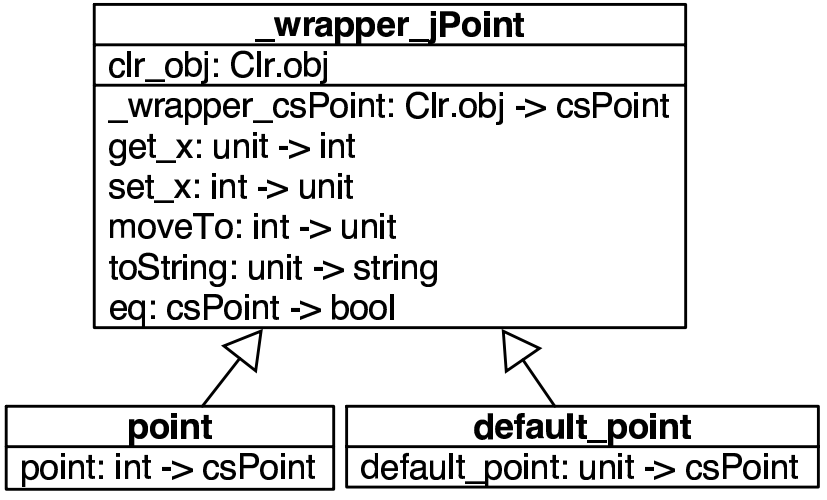
Abstract type `_clr_csPoint`

Object type `csPoint`

Wrapper `_wrapper_csPoint`

Users classes

`default_point`, `point`

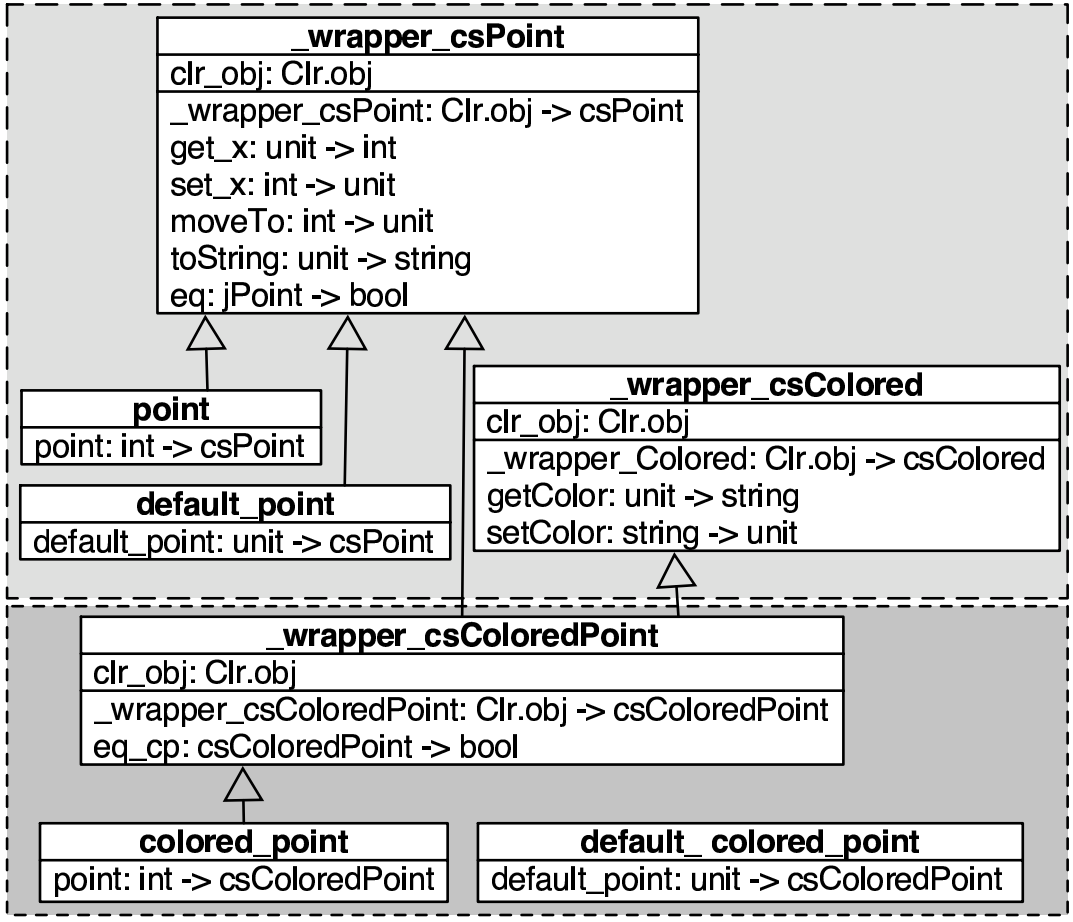


O'Jacaré.Net : Class ColoredPoint

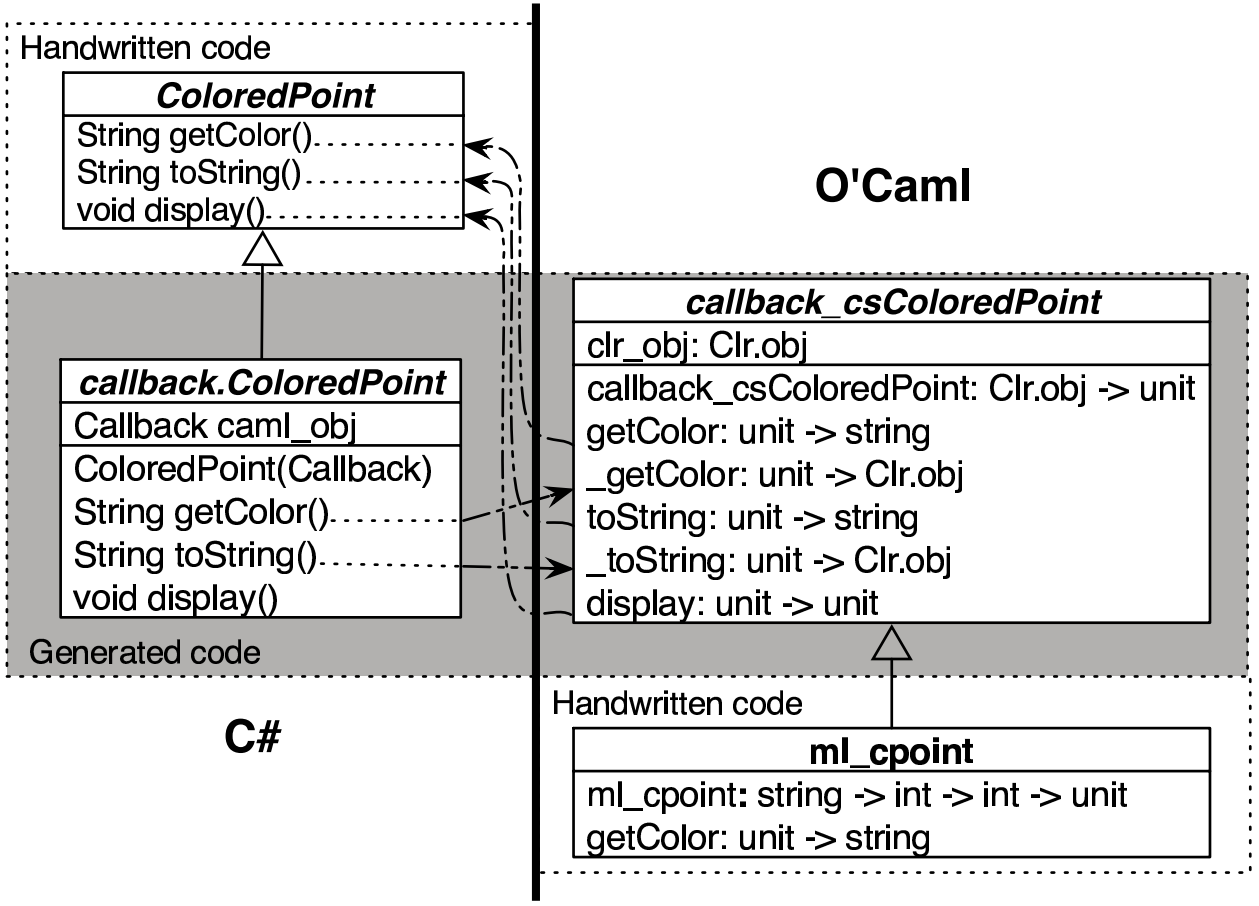
File `point.idl`

```
class ColoresPoint extends Point
    implements Colored {

    [name default_colored_point]
        <init> ();

    [name colored_point]
        <init> (int,string);

    [name eq_cp]
        boolean eq(ColoredPoint);
}
```

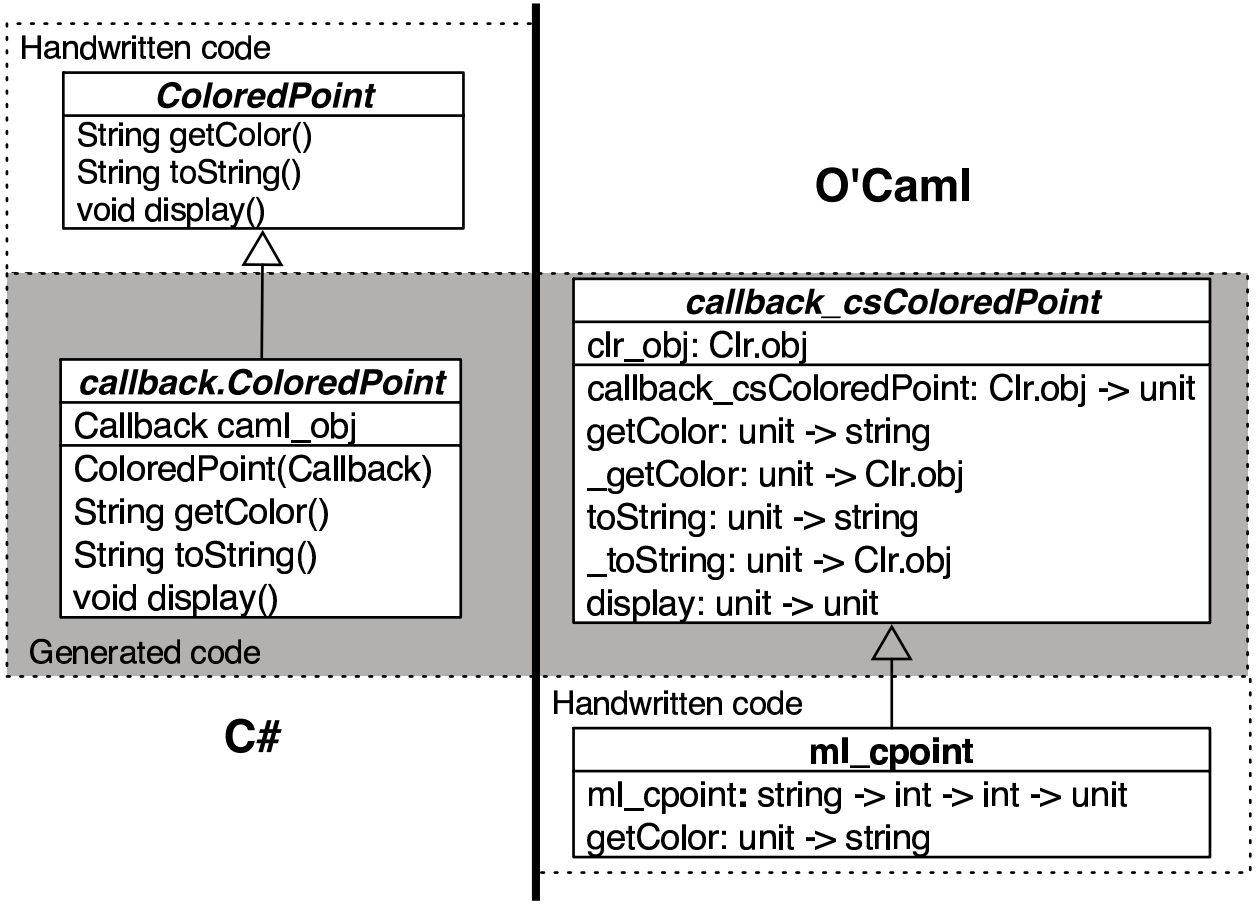
Generates : `point.ml`



O'Jacaré.Net : Callback attribute

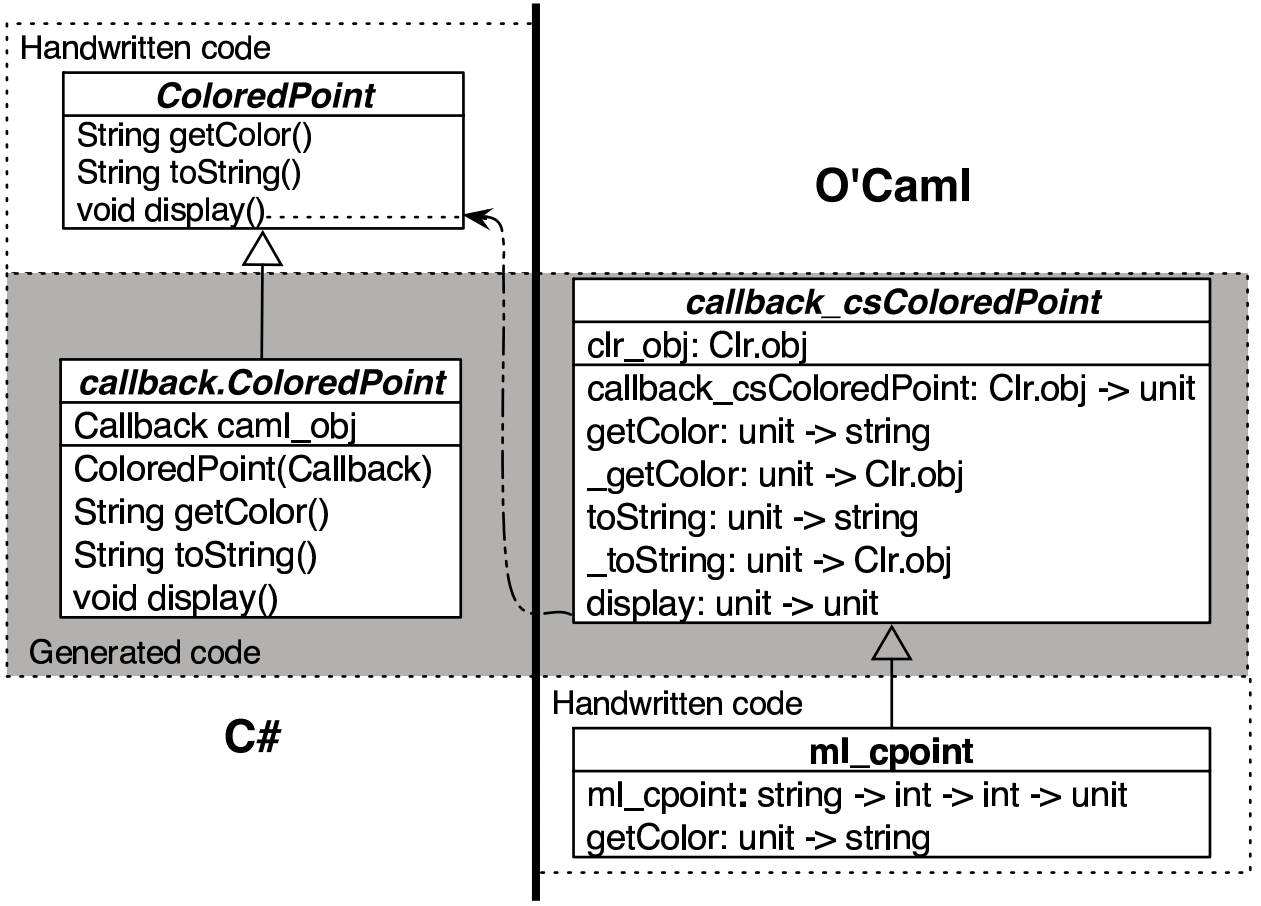


O'Jacaré.Net : Callback attribute



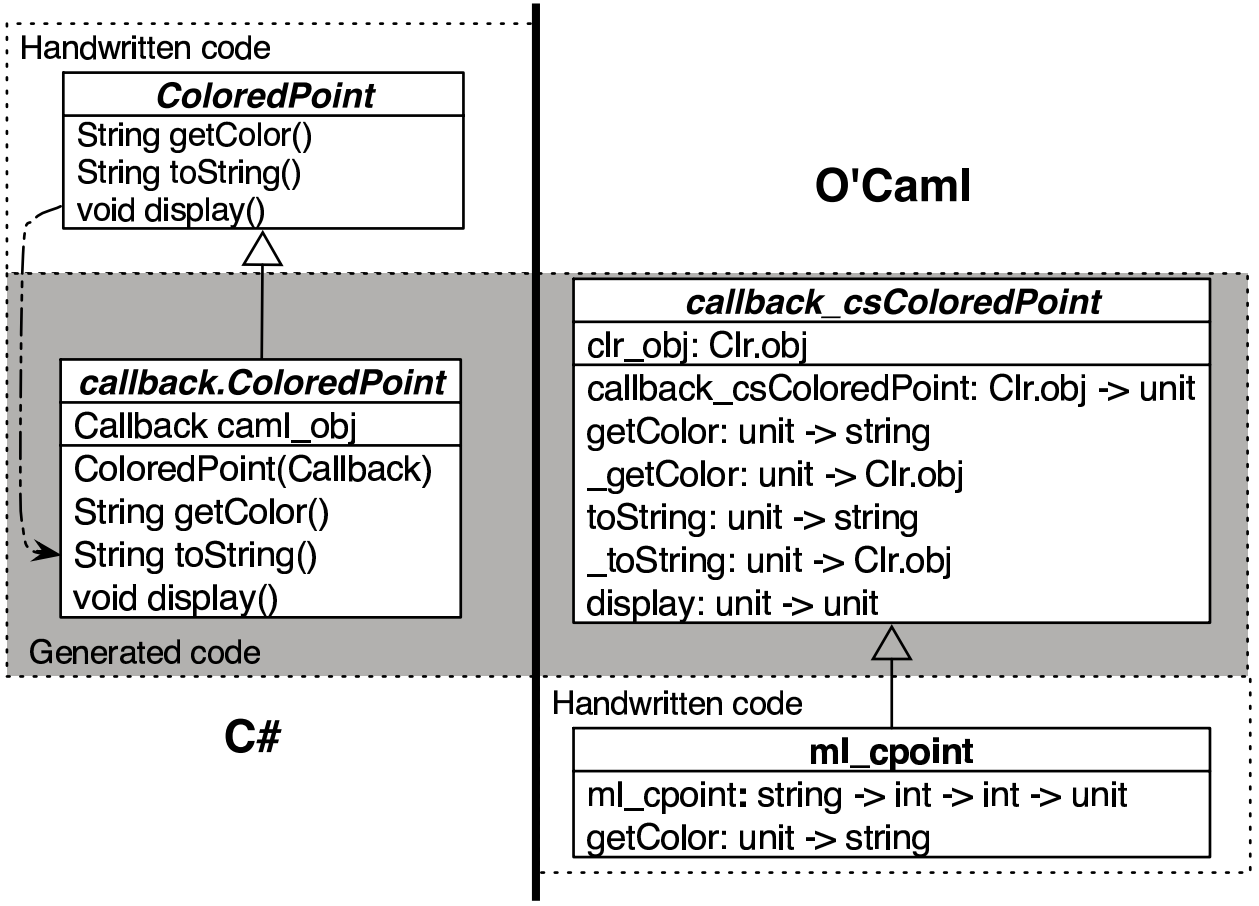

```
let mcp = new
 ml_cpoint
 "blue" 1
```

O'Jacaré.Net : Callback attribute



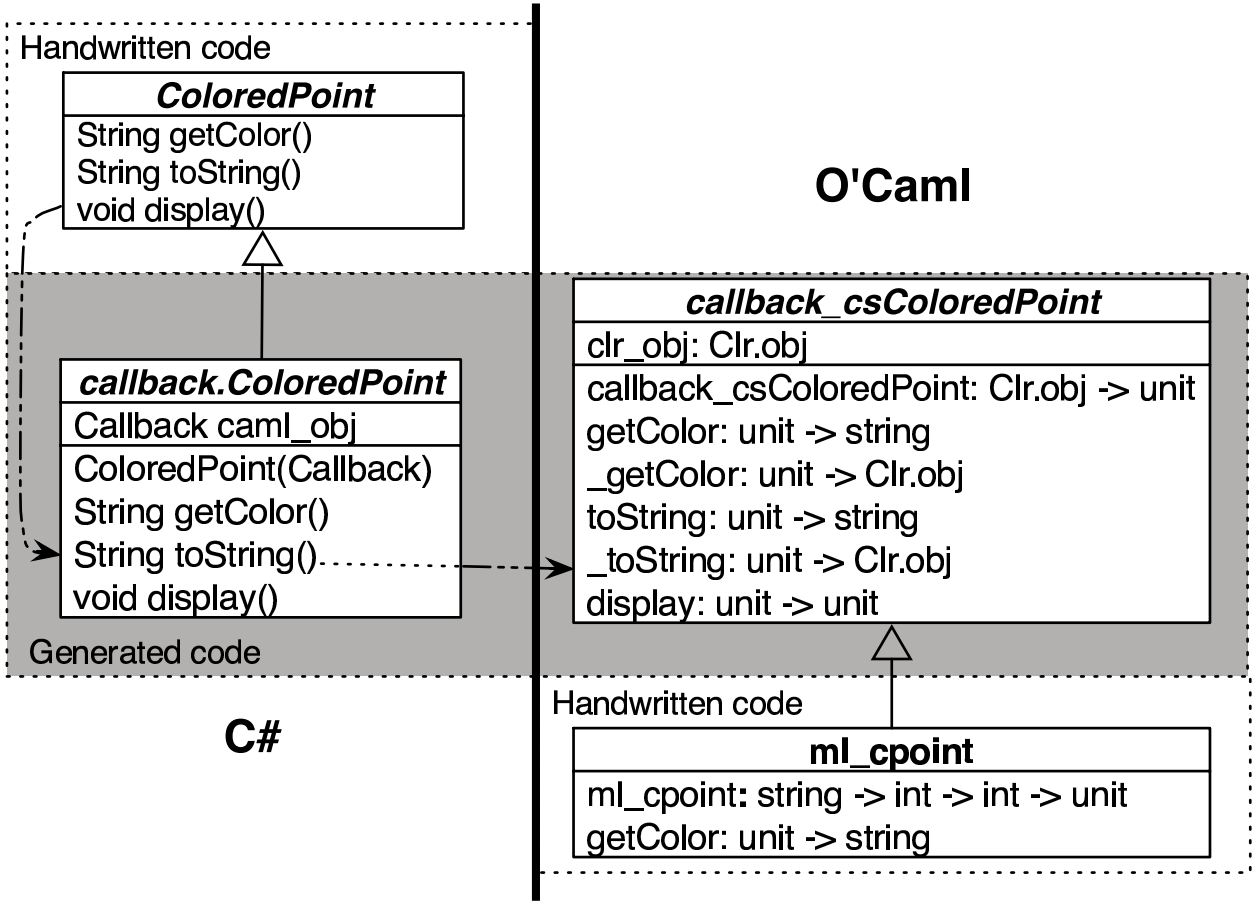

```
mcp#display()
```

O'Jacaré.Net : Callback attribute



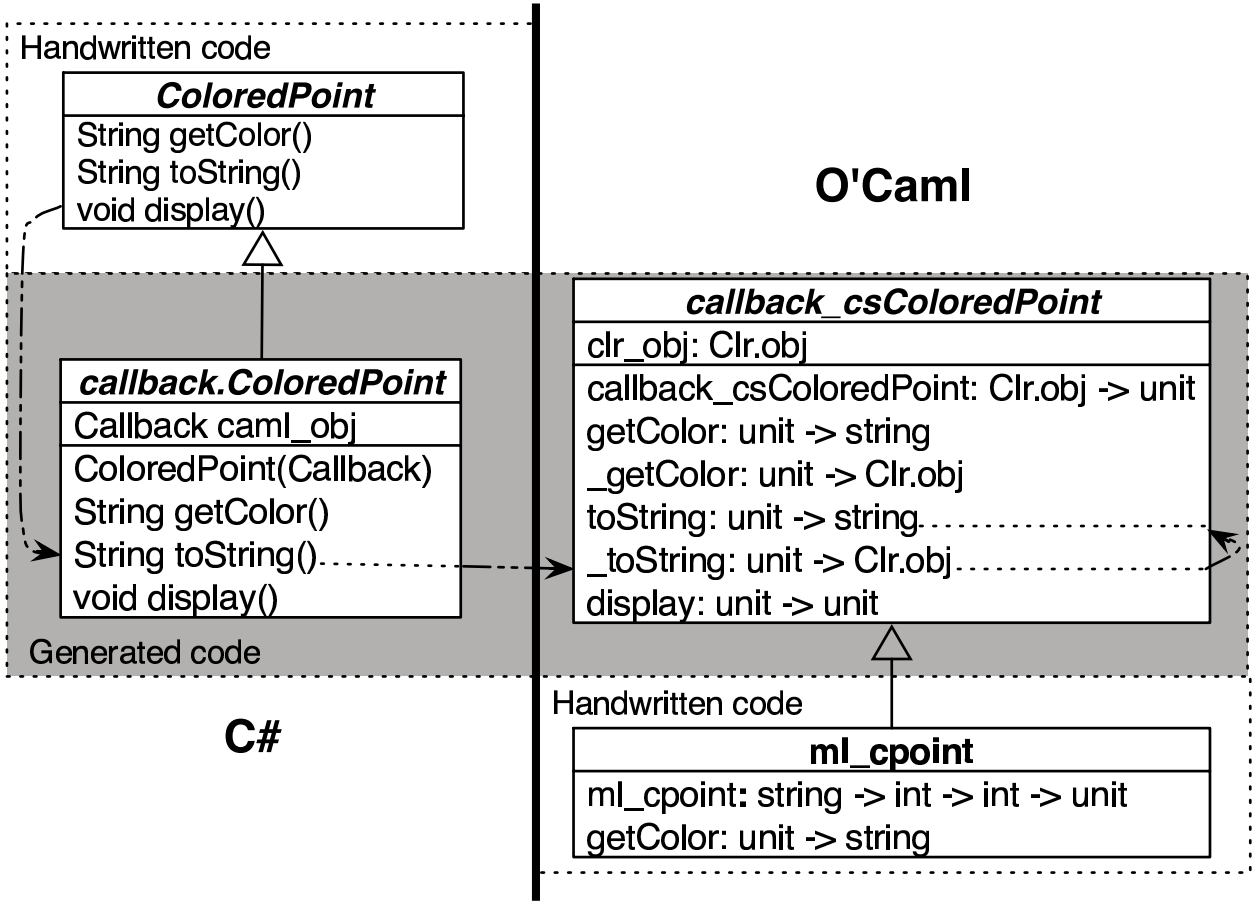

```
mcp#display()
this.toString()
```

O'Jacaré.Net : Callback attribute



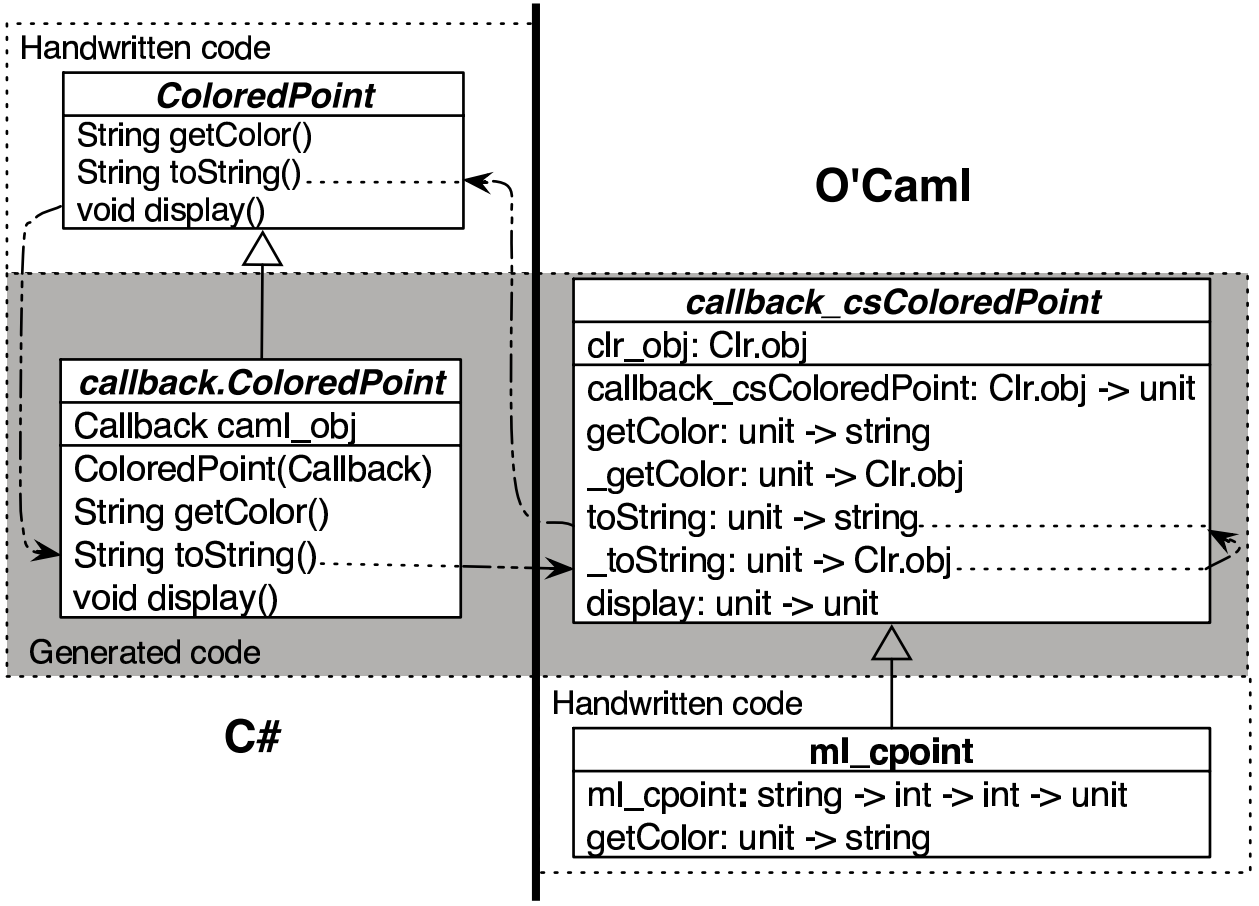

```
mcp#display()
this.toString()
```

O'Jacaré.Net : Callback attribute



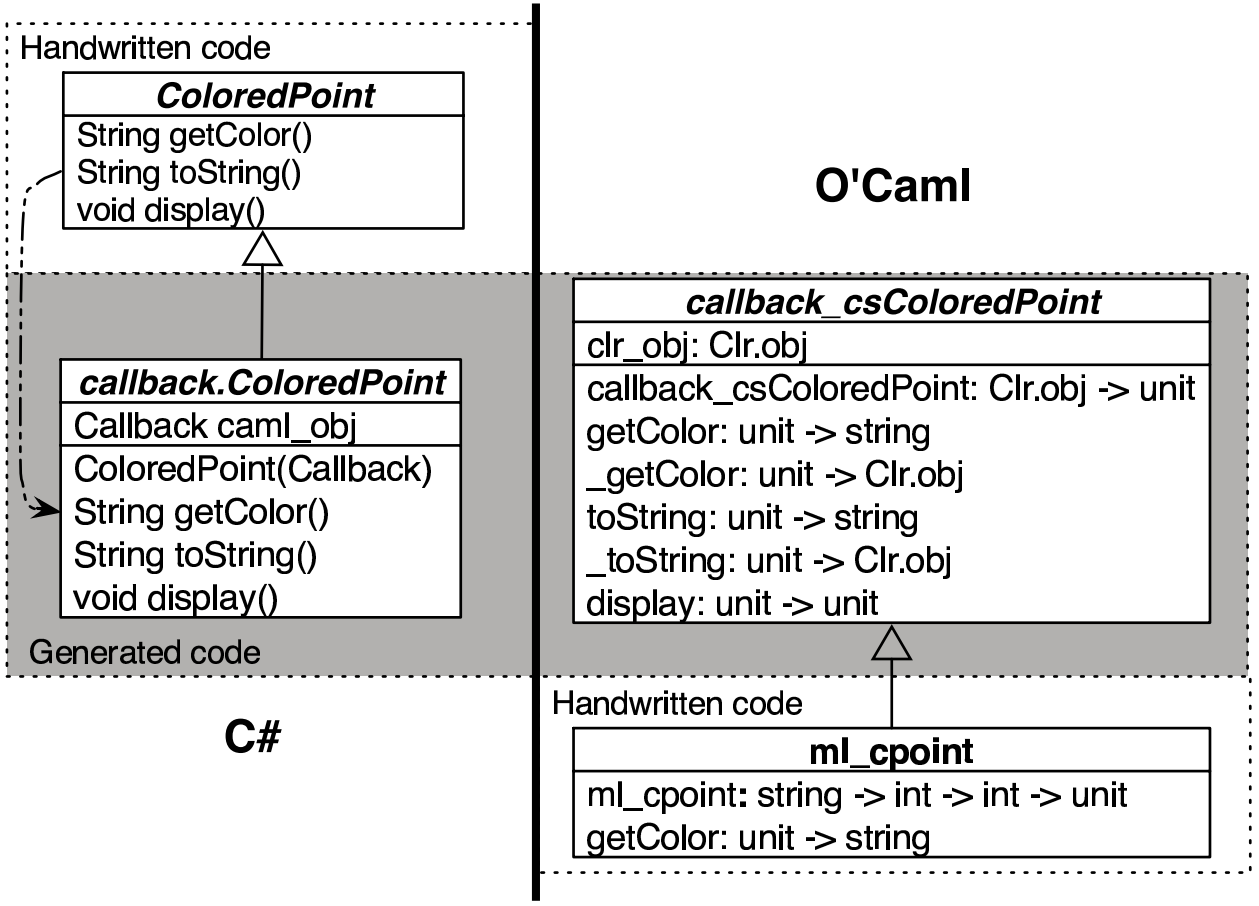

```
mcp#display()
this.toString()
```

O'Jacaré.Net : Callback attribute



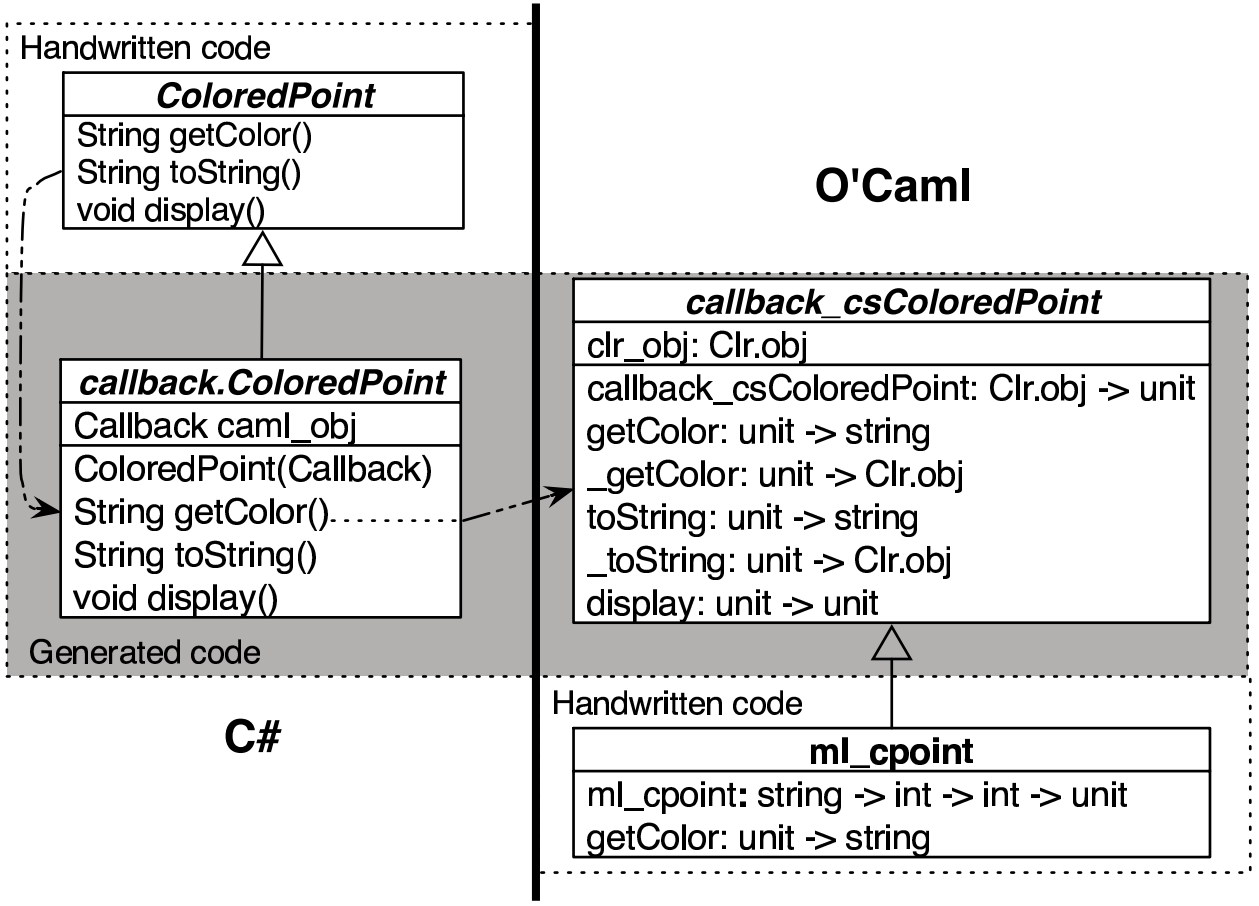

```
mcp#display()
this.toString()
```

O'Jacaré.Net : Callback attribute



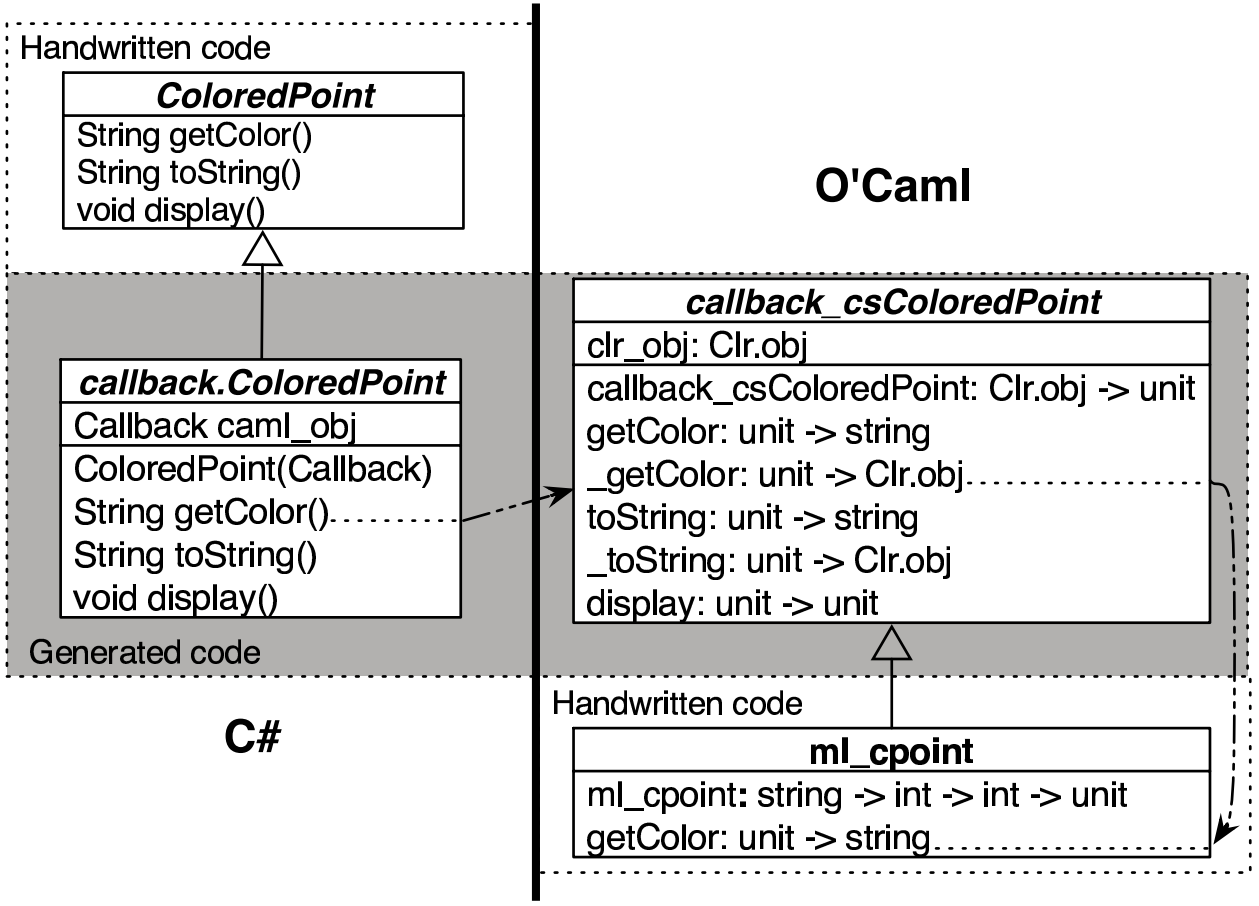

```
mcp#display()
this.toString()
this.getColor()
```

O'Jacaré.Net : Callback attribute



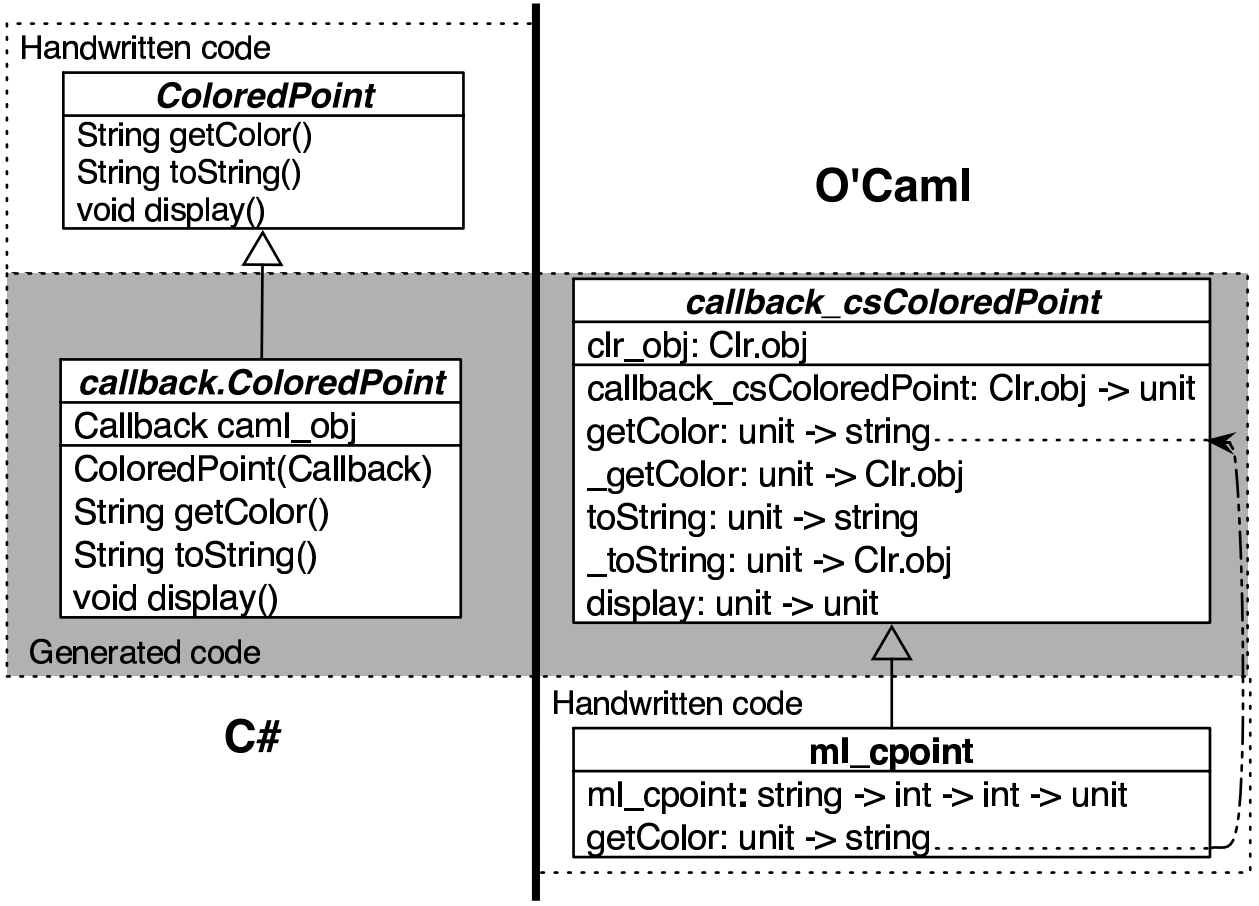

```
mcp#display()
this.toString()
this.getColor()
```

O'Jacaré.Net : Callback attribute



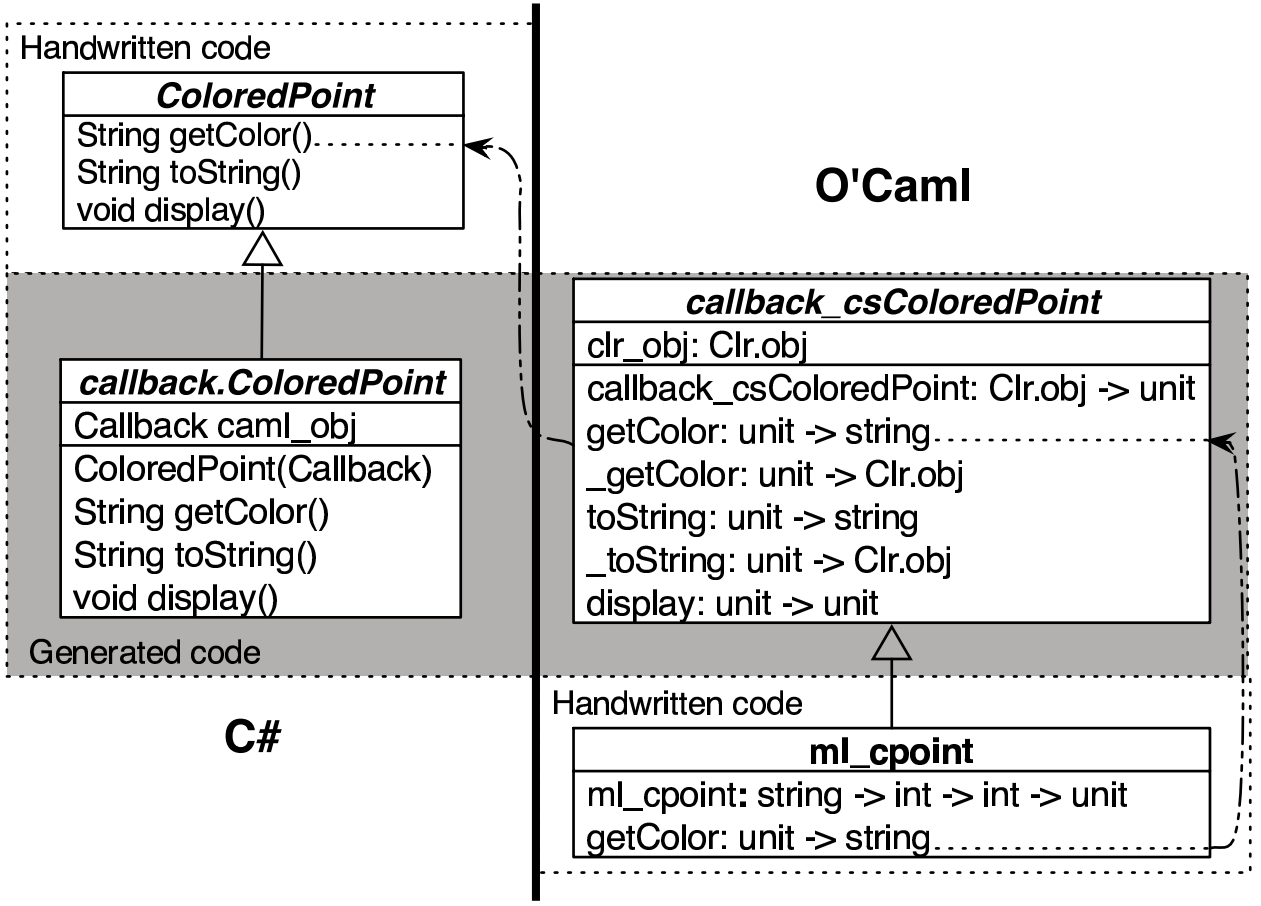

```
mcp#display()
this.toString()
this.getColor()
```

O'Jacaré.Net : Callback attribute



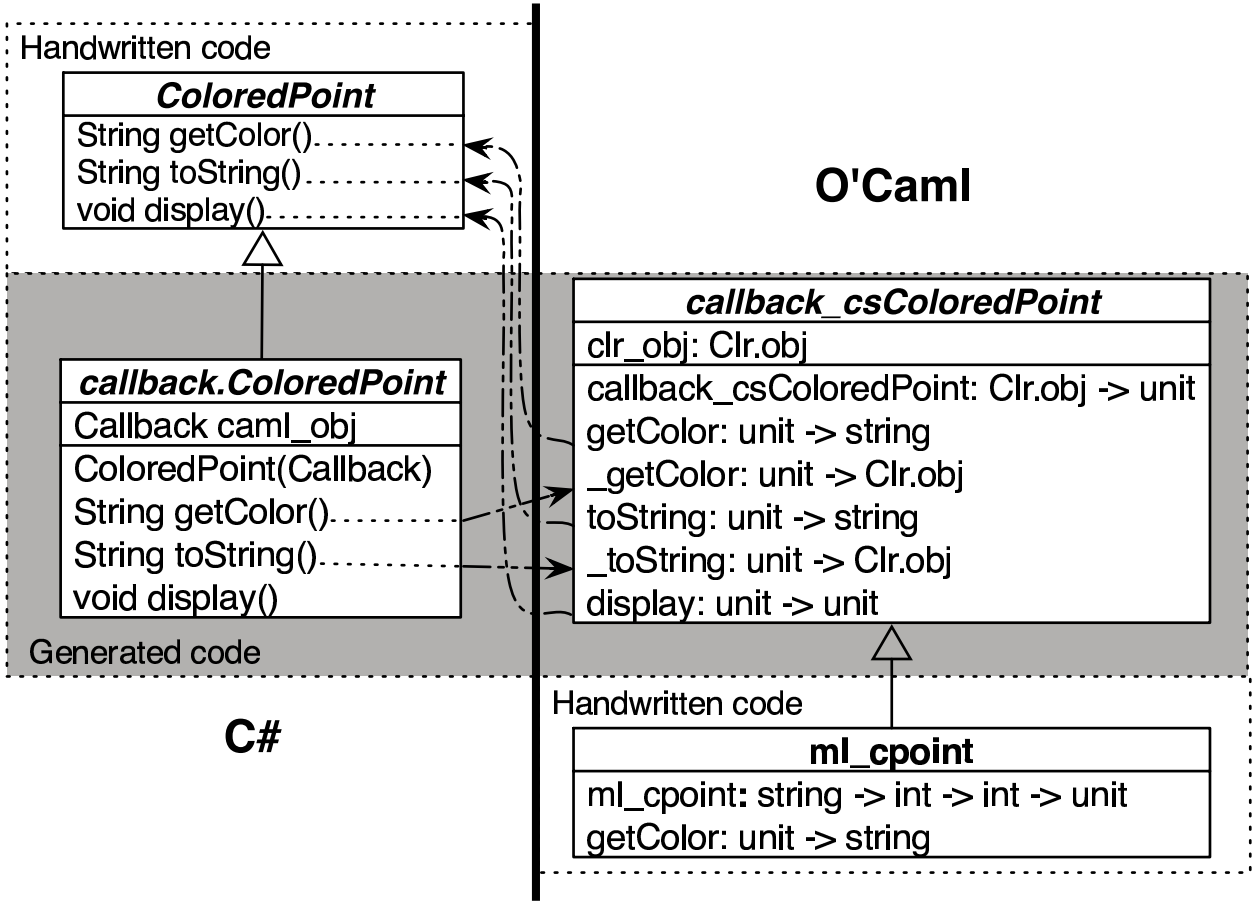

```
mcp#display()
this.toString()
this.getColor()
super#getColor()
```

O'Jacaré.Net : Callback attribute




```
mcp#display()
this.toString()
this.getColor()
super#getColor()
```

O'Jacaré.Net : Callback attribute




```
mcp#display()
this.toString()
this.getColor()
super#getColor()
```

Example - a Raytracer program 1/1

Two components:

A raytracer engine in O'Caml, in a class `Render`.

- It has a method `compute: *display* -> string -> unit`.
- Wishes to call a method `drawPixel` on object `display` (`drawPixel: int -> int -> int -> int -> int -> unit`).

A graphical interface in C# has a class `Display`.

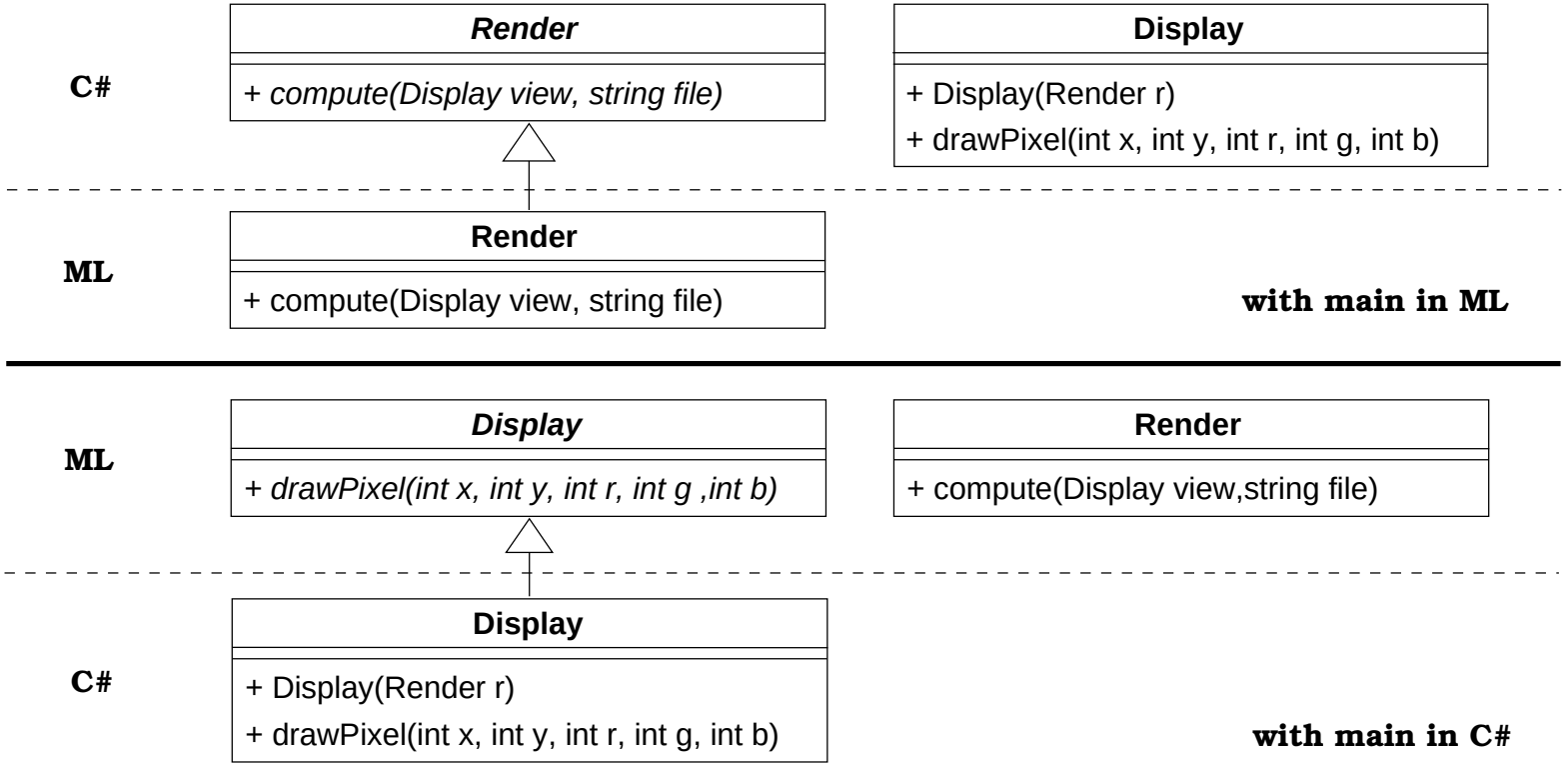
- with a `drawPixel` method:

  `void drawPixel (int x, int y, int r, int g, int b)`.
- A file dialog helps selecting a 3D scene, willing to call a `compute` method.

Communication is round tripping between the two components.

Example - a Raytracer program 2/2

This can be implemented with O'Jacaré.Net using cross-language late binding. Two solutions work:



Combining the two Objects Models

- Multiple inheritance of C# classes
- Downcasting C# objects in O'Caml

Multiple inheritance of C# classes

The file `rect.idl`

```
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 area();
}
```

The O'Caml program

```
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#area ());;
Printf.printf "toString=%s\n" (ggr#toString ());;
```

Downcasting C# objects in O'Caml

```
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>
```

- The generated O'Caml class hierarchy has root class `top`,
- O'Jacaré.Net defines type coercion functions from `top` to child classes.

Further work

Enhancements:

- Add delegation, genericity . . .
- Increase the IDL scope.

Applications:

- Embed functional programs,
- Promote O'Caml for specific fields of application (symbolic computations: parsers, compilers),
- Experiment new features in O'Caml (remoting . . .)