Objective Caml on .NET

The OCamlIL Compiler and Toplevel

Raphaël Montelatici\(^1\), Emmanuel Chailloux\(^2\) and Bruno Pagano\(^3\)

\(^1\) Equipe PPS (UMR 7126) - Université Denis Diderot (Paris VII)
\(^2\) Equipe PPS (UMR 7126) - Université Pierre et Marie Curie (Paris VI)
\(^3\) Esterel Technologies

.NET Technologies’2005 - Plzen - May, 2005
Summary

- Motivations, Constraints and Choice
- Objective Caml
- OCamlIL : Compiler and Toplevel
- O’Jacaré : Interoperability
- Related Work
- Current and Future Work
Motivations:

- To port O’Caml language and its libraries to the .NET platform

  and to be as compatible as possible with the standard (Inria implementation).

- To benefit from this platform: tools and interoperability
The OCamlIL project

Constraints:

- To have a good level of compatibility. (compatibility)
- To produce managed code. (safety)
- Don’t modify the original language. (compatible)
- To follow perpetual evolutions. (easy evolutions)
- To have some facilities to interoperate. (interoperability)
- And if possible, to prevent inefficiencies. (efficiency)
The OCamlIL project

Choice:
- OCamlIL is developed as a new back-end of the O’Caml compiler.

Benefits are:
- More accurate compatibility
- Easier to write
- Easier to maintain

Drawbacks are:
- Sticks to O’Caml implementation choices
- Needs a retyping step
- Entails efficiency penalties
Objective Caml (O’Caml)

- One of the most popular ML dialect:
  - efficient code,
  - large set of general purpose and domain specific libraries,
  - automatic memory management,
  - used both for teaching (academy) and for writing high-tech applications (industry).

- Product of research results since 80’s in: type theory, language design and implementation.

- Developed at INRIA (France). http://caml.inria.fr.
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).
- OO extension.
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).
- OO extension.
- Multi-paradigm (inside the same typing mechanism):
  - Object-oriented (class structuration),
  - SML-like parametric module.
O’Caml : Examples of type inference

**functional type:**

```ocaml
let compose f g = fun x -> f (g x);;
(\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta
```

**functional type over list:**

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

**object and functional type:**

```ocaml
let toStringNL o = o#toString() ^"\n";;
< toString : unit \to string ; .. > \to string
let h = map toStringNL;;
< toString : unit \to string; _.. > \to list \to
```

```
string list
```
type ulambda =
        Uvar of Ident.t
    | Ulet of Ident.t * ulambda * ulambda
    | Udirect_apply of function_label * ulambda list
    | Ugeneric_apply of ulambda * ulambda list
    | Uclosure of
        (function_label * int * Ident.t list * ulambda) list
        * ulambda list
    | Uoffset of ulambda * int
    | Uprim of primitive * ulambda list
    | ...

\(\lambda\)-calcul : explicit closures management, control structures
and primitives.
type primitive =
    | Pmakeblock of int
    | Pfield of int
    | Psetfield of int
    | Psequand | Psequor | Pnot
    | Pnegint | Paddint | Psubint | Pmulint | Pdivint | Pmodint
    | Pandint | Porint | Pxorint | Plslint | Plsrint | Pasrint
    | Pintcomp of comparison
    | Pintoffloat | Pfloatofint | Pnegfloat | Pabsfloat
    | Paddfloat | Psubfloat | Pmulfloat | Pdivfloat
    | Pfloatcomp of comparison
    | Pstringlength | Pstringrefu | Pstringsetu
    | Pstringrefs | Pstringsets
    | ...
Main Problem:
- Untyped intermediate language AND typed runtime
- Example: integers and blocks

Goals:
- To produce managed code without type errors
- To produce optimized code

Annotate primitives by types to ensure coherence.
2 approaches:
- To rebuild types
- OR to propagate types.
OCamlIL V1: type reconstruction

Minimal grammar for types:

\[ T ::= \text{int} \mid \text{block} \mid \text{string} \]
\[ \quad \mid \text{float} \mid \text{closure} \mid \text{unit} \mid \text{any} \]

- To discriminate kinds of blocks
- A different representation (than O’Caml) for sum types.

Implementation:

<table>
<thead>
<tr>
<th>OCaml</th>
<th>bool/int</th>
<th>float</th>
<th>string</th>
<th>unit</th>
<th>-&gt;</th>
<th>’a</th>
<th>others</th>
</tr>
</thead>
<tbody>
<tr>
<td>retyping</td>
<td>int</td>
<td>float</td>
<td>string</td>
<td>unit</td>
<td>closure</td>
<td>any</td>
<td>block</td>
</tr>
<tr>
<td>CTS</td>
<td>int32</td>
<td>float64</td>
<td>StringB</td>
<td>null/void</td>
<td>Closure</td>
<td>object</td>
<td>object[]</td>
</tr>
</tbody>
</table>
For sum types, we want an union type:

$$\text{sumtype} = \text{int} \cup \text{block}$$

**Problem:**

type \( t = \text{Zero} \mid \text{One} \mid \text{Node of} \ t \)

<table>
<thead>
<tr>
<th>O’Caml code</th>
<th>Clambda code</th>
</tr>
</thead>
</table>
| let cut = function   | let cut = closure(cut):
|                     | x -> if (isint x) then x
| Node n -> n         | else (field 0 x)                                                            |
| x -> x              |                                                                             |
| Type: \( t \rightarrow t \) | **Inferred type:** sumtype \( \rightarrow \) sumtype                      |
**OCamIL V1: type reconstruction**

```
type t = Zero | One | Node of t
```

<table>
<thead>
<tr>
<th>O’Caml code</th>
<th>Clambda code</th>
</tr>
</thead>
<tbody>
<tr>
<td>let hell a b =</td>
<td>let hell = closure(hell): a -&gt; b -&gt;</td>
</tr>
<tr>
<td>match a with</td>
<td>if (isint a) then</td>
</tr>
<tr>
<td></td>
<td>Zero -&gt; One</td>
</tr>
<tr>
<td></td>
<td>_ -&gt; b</td>
</tr>
<tr>
<td>Type: t -&gt; t -&gt; t</td>
<td>Inferred type: sumtype -&gt; int -&gt; int</td>
</tr>
</tbody>
</table>

The following expression produces an incoherence:

```
hell One (Node Zero)
```
OCamlIL V1 : type reconstruction

Solution: to modify sum types representation.

- To represent all constructors (including constants) by blocs.
- More elegant solutions don’t work (because polymorphic variants)
- Une solution plus fine ne marcherait pas (à cause des variants polymorphes)

It is a first modification before our back-end.
Other Main Implementation Choices

Uniform Representation:
- All values can be represented by an `Object` boxing/unboxing for scalar values

Closures:
- Each abstraction (`fun`) becomes a class,
  - sub-class of the abstract class `Closure`.
  - A closure is an instance of their corresponding class.

Application:
- direct apply
- using a general mechanism:
  - which can create a new closure (with a richer environment)
  - or can execute the real code (body of abstraction)
Exceptions:

- sub-classes of .NET exceptions

Objects:

- only records (instance variables and virtual method table)
- general application:
  
  \[ \text{\texttt{\textit{o.m a1 a2}} \implies \text{\texttt{GET(o,m) o a1 a2}}} \]

Functors:

- represented by closures too
Maturity

Libraries :
- standard library
- Graphics
- Threads
- Dynlink

Compatibility :
- Bootstrapped compiler
- Toplevel
Bootstrap

Building Steps

Bootstrapping Steps
Toplevel

BCL

Reflection

Toplevel

Toplevel Engine

SymTable

ocamiltop Application Domain

Output

Disk

phrase_1 ...

phrase_n

Input

1 2 2 3 4 5 a 5 b 6

phrase_1 ...

phrase_n

(5c)
Interoperability

low-level FFI, using static methods .NET.

external il_getenv: string -> string = "string" "System.Environment::GetEnvironmentVariable" "string"

;;

let getenv var = match il_getenv var with
    | "" -> raise Not_found
    | s -> s

;;

Example : implementation of Sys.getenv
Interoperability : Low Level FFI

```ocaml
(BSCAML) Objective Caml version 3.06+camil
# let zodiac = List.map (fun i -> String.make 1 (char_of_int i))
[0x9f20,0x725b;0x864e;0x5154;0x9f99;0x86c7;0x9a6c;0x7f8a;0x7334;0x9e21;0x72d7;0x732a];;
# List.sort String.compare zodiac;;
: String.t list =["兔", "牛", "狗", "猪", "猴", "羊", "虎", "蛇", "马", "鸡", "鼠", "龙"]
# type culture_info;;
type culture_info
# external create_culture:string -> culture_info = "class System.Globalization.CultureInfo"
"System.Globalization.CultureInfo" "CreateSpecificCulture" "string";;
external create_culture : string -> culture_info = "CreateSpecificCulture" "CreateSpecificCulture";;
# external uni_compare:string -> string -> bool -> culture_info -> int = "int" "System.String"
"Compare" "string" "string" "bool" "class System.Globalization.CultureInfo";;
external uni_compare : string -> string -> bool -> culture_info -> int = "Compare" "Compare"
# let pinyin_compare s1 s2 =
    let chinese = create_culture "zh-CN" in
    uni_compare s1 s2 true chinese;;
val pinyin_compare : string -> string -> int = <fun>
# List.sort pinyin_compare zodiac;;
: String.t list =["狗", "猴", "虎", "鸡", "龙", "马", "牛", "蛇", "鼠", "兔", "羊", "猪"]
#
# Interoperability

<table>
<thead>
<tr>
<th>Features</th>
<th>C#</th>
<th>O’Caml</th>
<th>Features</th>
<th>C#</th>
<th>O’Caml</th>
</tr>
</thead>
<tbody>
<tr>
<td>classes</td>
<td>✓</td>
<td>✓</td>
<td>inheritance ≡ sub-typing?</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>late binding</td>
<td>✓</td>
<td>✓</td>
<td>overloading</td>
<td>✓</td>
<td>*</td>
</tr>
<tr>
<td>early binding</td>
<td>✓</td>
<td>*</td>
<td>multiple inheritance</td>
<td>*</td>
<td>✓</td>
</tr>
<tr>
<td>static typing</td>
<td>✓</td>
<td>✓</td>
<td>parametric classes</td>
<td>*</td>
<td>✓</td>
</tr>
<tr>
<td>dynamic typing</td>
<td>✓</td>
<td>*</td>
<td>packages/modules</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>sub-typing</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**IDL**: an intersection of the two models: for which inheritance and subtyping relations are equivalent, overloading and binary methods are not allowed. No multiple inheritance nor parametric classes.
O’Jacaré.Net : Class Point

File `point.idl`

```plaintext
class Point {
    int x;

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

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

Generates: `point.ml`

Object type `csPoint`

Wrapper `_wrapper_csPoint`

Users classes

```
default_point, point
```

```
_wrapped_jPoint

clr_obj: Clr.obj

_wrapped_csPoint: Clr.obj -> csPoint
get_x: unit -> int
set_x: int -> unit
moveTo: int -> unit
toString: unit -> string
eq: csPoint -> bool
```

```
point
default_point

point: int -> csPoint
default_point: unit -> csPoint
```
<table>
<thead>
<tr>
<th>File</th>
<th>point.idl</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>ColoredPoint extends Point</td>
</tr>
<tr>
<td>implements</td>
<td>Colored</td>
</tr>
<tr>
<td>[name] default_colored_point</td>
<td></td>
</tr>
<tr>
<td>&lt;init&gt;</td>
<td>()</td>
</tr>
<tr>
<td>[name] colored_point</td>
<td></td>
</tr>
<tr>
<td>&lt;init&gt;</td>
<td>(int,string)</td>
</tr>
<tr>
<td>[name] eq_cp</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>eq(ColoredPoint);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generates</th>
<th>point.ml</th>
</tr>
</thead>
<tbody>
<tr>
<td>_wrapper_csPoint</td>
<td></td>
</tr>
<tr>
<td>clr_obj: Clr.obj</td>
<td></td>
</tr>
<tr>
<td>_wrapper_csPoint: Clr.obj -&gt; csPoint</td>
<td></td>
</tr>
<tr>
<td>get_x: unit -&gt; int</td>
<td></td>
</tr>
<tr>
<td>set_x: int -&gt; unit</td>
<td></td>
</tr>
<tr>
<td>moveTo: int -&gt; unit</td>
<td></td>
</tr>
<tr>
<td>toString: unit -&gt; string</td>
<td></td>
</tr>
<tr>
<td>eq: jPoint -&gt; bool</td>
<td></td>
</tr>
</tbody>
</table>

| point |
| clr_obj: Clr.obj |
| point: int -> csPoint |

| _wrapper_csColored |
| clr_obj: Clr.obj |
| _wrapper_Colored: Clr.obj -> csColored |
| getColor: unit -> string |
| setColor: string -> unit |

| default_point |
| clr_obj: Clr.obj |
| default_point: unit -> csPoint |

| _wrapper_csColoredPoint |
| clr_obj: Clr.obj |
| _wrapper_csColoredPoint: Clr.obj -> csColoredPoint |
| eq_cp: csColoredPoint -> bool |

| colored_point |
| point: int -> csColoredPoint |

| default_colored_point |
| default_point: unit -> csColoredPoint |
Objective Caml on .NET – p.27/43
Combining the two Objects Models

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

<table>
<thead>
<tr>
<th>The file rect.idl</th>
<th>The O'Caml program</th>
</tr>
</thead>
<tbody>
<tr>
<td>package mypack;</td>
<td>open Rect;;</td>
</tr>
<tr>
<td>class Point {</td>
<td>class geom_graph_rect p1 p2 =</td>
</tr>
<tr>
<td>[name point] &lt;init&gt; (int, int);</td>
<td>object</td>
</tr>
<tr>
<td>}</td>
<td>inherit geom_rect p1 p2 as super_geo</td>
</tr>
<tr>
<td>class GraphRectangle {</td>
<td>inherit graph_rect p1 p2 as super_graph</td>
</tr>
<tr>
<td>[name graph_rect] &lt;init&gt;(Point, Point);</td>
<td>end;;</td>
</tr>
<tr>
<td>string toString();</td>
<td>let p1 = new point 10 10;;</td>
</tr>
<tr>
<td>}</td>
<td>let p2 = new point 20 20;;</td>
</tr>
<tr>
<td>class GeomRectangle {</td>
<td>let ggr = new geom_graph_rect p1 p2;;</td>
</tr>
<tr>
<td>[name geom_rect] &lt;init&gt;(Point, Point);</td>
<td>Printf.printf &quot;area=%g\n&quot; (ggr#area());;;</td>
</tr>
<tr>
<td>double area();</td>
<td>Printf.printf &quot;toString=%s\n&quot; (ggr#toString());;;</td>
</tr>
</tbody>
</table>
The generated O’Caml class hierarchy has root class \texttt{top},

O’Jacaré.Net defines type coercion functions from \texttt{top} to child classes.
Interoperability
Interoperability

OCamlIL Web Control

Click on window to close.
Related Work

- **F#**
  - An Caml-Light Core: no functors, no O’Caml objects.
  - Nombreuses autres petites incompatibilités
  - Well integrated C# object model (not the O’Caml model)
  - no toplevel
  - similar efficiency

- **SML.NET**
  - compatibility with SML
  - integration of a C# syntax
  - no toplevel
  - better efficiency (global monomorphisation)
  - SML ≠ O’Caml...
Current Work

to propagate types from types syntax tree:

- introducing a typed intermediate language

interest:

- allows to represent more precisely O’Caml types
- better efficiency and easier debug

difficulty:

- propagation during all optimization steps (pattern matching, . . .)
OCamIL V2: type propagation

OCaml Bytecode Instructions
- ocamlc

Lambda Code
- nymp Code
- Mach Code
- Native Code
- Tlambda Code
- Clambda Code
- Cmm Code

Typed Syntax Tree

standard INRIA compilers
- ocamlopt

OCamIL project
- ocaml
Type information is available in syntax tree, but is not propagating in intermediate languages.

type t = A of int * t | B of int

let f = function
  A(0,u) , B i -> A(i,u)
| ( A(x,_) , A(y,_) | B x , B y ) -> B(x+y)
| A(_,u) , _ -> u
| _ , _ -> B 0
param ->
catch 1 x' y'

let left = (field 0 param) in

switch left
  0 -> let x = (field 0 left) in
      catch 3
      if (x != 0) then <jump 3> else
      let right0 = (field 1 param) in
      switch right0
      0 -> <jump 3>
      1 -> [makeblock 0 (field 0 right0) (field 1 left)]
      c-with(3) let right1 = (field 1 param) in
      switch right1
      0 -> <jump 1 x (field 0 right1)>
      1 -> (field 1 left)
  1 -> let right = (field 1 param) in
      switch right
      0 -> [1: 0]
      1 -> <jump 1 (field 0 left) (field 0 right)>
      c-with(1) [makeblock 1 (x'+y')]
OCamlIL V2: type propagation

```
param:t * t ->
catch 1 (x':int) (y':int)
  let left:t = (field 0 param) in
  switch left
  0 -> let x:int = (field 0 left) in
      catch 3
      if (x != (0:int)):bool then <jump 3>:t else
      let right0:t = (field 1 param) in
      switch right0
      0 -> <jump 3>:t
      1 -> [makeblock 0 (field 0 right0):int
              (field 1 left):t]:t
  c-with(3) let right1:t = (field 1 param) in
  switch right1
  0 -> <jump 1 x (field 0 right1):int>:t
  1 -> (field 1 left):t
  1 -> let right:t = (field 1 param) in
  switch right
  0 -> [1: 0]:t
  1 -> <jump 1 (field 0 left):int (field 0 right):int>:t
  c-with(1) [makeblock 1 (x'+y'):int]:t
```
OCamlIL V2: type propagation

```ocaml
param:t*t ->
catch 1 (x’:int) (y’:int)
  let left:t = (field 0 param) in
  switch left
    0 -> let x:int = (field 0 left:t.A) in
        catch 3
        if (x != (0:int)):bool then <jump 3>:t else
        let right0:t = (field 1 param) in
        switch right0
          0 -> <jump 3>:t
          1 -> [makeblock 0 (field 0 right0:t.B):int
                          (field 1 left:t.A):t]:t
  c-with(3) let right1:t = (field 1 param) in
  switch right1
    0 -> <jump 1 x (field 0 right1:t.A):int>:t
    1 -> (field 1 left:t.A):t
1 -> let right:t = (field 1 param) in
  switch right
    0 -> [1: 0]:t
    1 -> <jump 1 (field 0 left:t.A):int (field 0 right:t.B):int>
  c-with(1) [makeblock 1 (x’+y’):int]:t
```
OCamIL V2 : type propagation

Grammar for retyping:

<table>
<thead>
<tr>
<th>O’Caml</th>
<th>type ‘a r = {x:’a; y:int; z:('a*’a) r}</th>
</tr>
</thead>
<tbody>
<tr>
<td>retypage</td>
<td>record(r, x:any, y:int, z:r)</td>
</tr>
<tr>
<td>CTS</td>
<td>class r {object x; int y; r z;}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>O’Caml</th>
<th>type t = A of t * t</th>
<th>B of int</th>
</tr>
</thead>
<tbody>
<tr>
<td>retypage</td>
<td>sumtype(t, t.A, t.B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>record(t.A, x0:t, x1:t)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>record(t.B, x0:int)</td>
<td></td>
</tr>
<tr>
<td>CTS</td>
<td>class t {int get_tag();}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>class t_A inherits t {t x0; t x1;}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>class t_B inherits t {int x0;}</td>
<td></td>
</tr>
</tbody>
</table>
OCamIL V2 : type propagation

- More precise types for primitives and best representation for sum types and records.
- More difficult to implement
- Deeper modifications of the O’Caml compiler
Benchmarks

The following benchmarks ran on a Windows XP Pentium IV 2.4GHz station. They are designed to run in about a second under the native O’Caml compiler (*ocamlopt*).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ocamlopt</th>
<th>ocamlc</th>
<th>OCamlLR</th>
<th>OCamlLP</th>
<th>F#</th>
<th>sml.net</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boyer</td>
<td>1.45</td>
<td>5.21</td>
<td>55.8</td>
<td>50.4</td>
<td>52.1</td>
<td>36.8</td>
</tr>
<tr>
<td>KBGr</td>
<td>1.14</td>
<td>1.31</td>
<td>11.9</td>
<td>11.7</td>
<td>11.8</td>
<td>6.18</td>
</tr>
<tr>
<td>KBGeo</td>
<td>1.34</td>
<td>2.75</td>
<td>59.3</td>
<td>59.2</td>
<td>66.1</td>
<td>28.5</td>
</tr>
<tr>
<td>Nucleic</td>
<td>0.81</td>
<td>5.82</td>
<td>11.5</td>
<td>7.62</td>
<td>7.48</td>
<td>0.81</td>
</tr>
</tbody>
</table>

Performance tests (real time in seconds).
Future Work

- To complete the types propagation version, and to compare both
- To implement some optimizations (closures, exceptions) and to use generic IL.
- To build a debugger that explores O’Caml values
- To parameterize backend to produce other byte-codes