Objective Caml on .NET: The OCamIL Compiler and Toplevel

Raphaël Montelatici [*]

Emmanuel Chailloux [†] Bruno Pagano [‡]

ABSTRACT

We present the OCamIL compiler for Objective Caml that targets .NET. Our experiment consists of adding a new back-end to the INRIA Objective Caml compiler that generates CIL bytecode. Among all the advantages of code reuse, ensuring compatibility while keeping all the expressiveness of the original language is particularly interesting. This allowed us to bootstrap the OCamIL compiler as a .NET component and build an interactive loop (toplevel) which may be embedded within .NET applications. This work deals with typing issues because OCamIL needs to translate an untyped intermediate language into a typed bytecode. We discuss various intermediate language retyping techniques and their consequences on performances. We also present applications of interoperability of Objective Caml and C# components.

1. INTRODUCTION

The .NET [1] platform is often presented as a universal framework that can host software components developed in numerous languages. It offers a *Common Type System* (CTS) and a runtime environment CLR (*Common Language Runtime*) built on a bytecode machine. By assuming compliance to the CTS type system, components interoperate safely. This has motivated the adaptation of various languages, such as C#, J#, A#, Eiffel, Scheme, Sml, F#, P# or Mercury.

Even though the main implementation of .NET runs on Windows, some Open Source projects provide implementations for BSD Unix and Windows (Rotor [2] and Linux (Mono [3]). That reminds of Java's motto: "Compile once, run everywhere". There is a hope for a safe and efficient multi-language platform with a single runtime, running on numerous systems. We experiment the integration of a full-fledged functional language in this environment by writing a .NET compiler for the INRIA Objective Caml [4] (thereafter shortened as O'Caml).

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. Its implementation includes a bytecode and a native code compiler, which generates efficient programs.

OCamIL [5] is a project which aims at compiling O'Caml to the .NET environment. We believe it can help make popular O'Caml applications. Our primary goals are compatibility with O'Caml and interoperability.

In order to help compliance with the original language, OCamIL is developed as a new back-end of the O'Caml compiler. This approach quickly succeeds in producing a full-fledged compiler for the whole language. We achieve bootstrapping as a sizeable compatibility test. Taking advantage of the .NET reflection API, OCamIL can dynamically emit code and execute it, which is a useful feature to build a toplevel interaction loop. Both compiler and toplevel can be redistributed as .NET components. The main part of O'Caml standard library and the O'Caml `graphics`, `threads` and `dynlink` libraries have been ported. Func-

[*] Equipe Preuves, Programmes et Systèmes (UMR 7126)
Université Denis Diderot (Paris 7)
2 place Jussieu, 75005 Paris, France
Email:Raphael.Montelatici@pps.jussieu.fr

[†] Equipe Preuves, Programmes et Systèmes (UMR 7126)
Université Pierre et Marie Curie (Paris 6)
4 place Jussieu, 75005 Paris, France.
Email: Emmanuel.Chailloux@pps.jussieu.fr

[‡] Esterel technologies,
679 Av Julien Lefèbvre, 06270, Villeneuve- Loubet, France
Email: Bruno.Pagano@esterel-technologies.com

.NET Technologies'2005 conference proceedings, ISBN 80-86943-01-1

tional, imperative and object-oriented features are implemented, as well as the module system (functors, modular compilation).

Interoperability is achieved using a two-layered technique: a low-level unsafe foreign function interface supports a high-level interfacing through O'Caml objects using an IDL approach.

We first present the relevant features of the .NET platform from a compiler writer's point of view, then give an outline of the OCamIL implementation and describe the building of the toplevel interactive loop from the bootstrapped compiler. We then expose the principles of OCamIL interoperability and give examples of applications. We finally discuss related work and outline future work.

2. THE .NET PLATFORM

The .NET Common Language Runtime consists of a typed stack-based bytecode called CIL (*Common Intermediate Language*), an execution system and a support library BCL (*Base Class Library*). Let us enumerate some features of the .NET platform for Windows developped by Microsoft:

The type system is designed around an object model featuring single inheritance, Java-style interfaces and exceptions. In addition to Reference Types (for heap-allocated objects), it supports stack-allocated Value Types (which range from basic types to complex structured types). Dedicated bytecode instructions (`box` and `unbox`) switch between the two kinds of representation. The type system is geared towards dynamic management: it supports run-time type tests, checked coercions and reflection capabilities.

Safety is based on typing. Verification rules are implemented in the runtime, tracking down stack inconsistencies and dependencies resolving errors (for instance erroneous calls to foreign methods). The CIL bytecode conforming to typing and verification constraints is called "managed code". Unmanaged code gives access to unsafe languages like C++. The runtime environment also features a Garbage Collection mechanism, which frees the developer from memory management issues.

Deployment: The fundamental .NET component is called an *assembly*: it is a self-contained unit of deployment. Assemblies can be signed with a cryptographic key so that the hosting computer can trust the embedded code: this allows sharing a piece of software by installing the assembly in the GAC (*Global Assembly Cache*), a special assembly repository. This helps versioning and localization management altogether.

Performances: The execution relies on a systematic Just In Time compilation mechanism (each method is compiled to native code at first call). It is possible to bypass this behavior by pre-compiling an assembly to a native image.

The CLR provides useful features for functional languages implementations, such as tail calls. However, closures, which are ubiquitous data structures in functional languages, are not supported natively by the CLR. The ILX extension [6] is developed to address this issue. Parametric polymorphism is also hard to implement efficiently, but change might be on its way with the possible addition of Generics [7, 8] to the forthcoming release of the CLR.

3. THE O'Caml LANGUAGE

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 [9]. One key feature of the 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 can.

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

Some of the most important characteristics of the O'Caml object model are:

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

4. THE OCamIL COMPILER

Our main goal is to port O'Caml to the .NET platform and be as compatible as possible with the standard INRIA implementation. Granting priority to compliance is not an easy task because the O'Caml language is perpetually evolving: new versions of the standard compiler are released on a regular basis, yielding major additions to the language. We choose to implement OCamIL as a back-end to the standard compiler, in order to reuse as much code as possible and later on to prevent tiresome modifications when upgrading to new O'Caml versions.

To be more precise: parsing, typing and first code transformations are left to the standard O'Caml compiler. Our back-end gets the internal representation `Clambda`[1] from the compiler front-end, as sketched in figure 1. At that stage, several code transformations have been realised. Further steps on the `ocamlopt` branch, which specialize code for specific processor architectures, are useless to OCamIL.

We introduce a new intermediate representation called `Tlambda`, the purpose of which is discussed in the following sections.

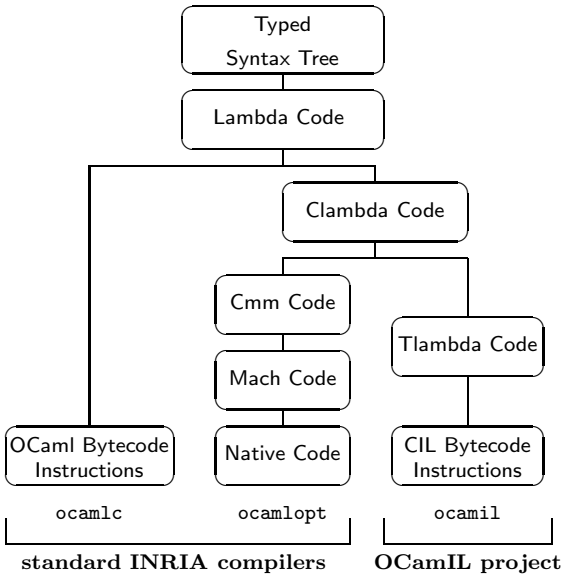

Fig. 1: OCamIL inside O'Caml.

[1]With respect to the `Lambda` code which handles functional values, `Clambda` explicitly manages closures and implements direct application.

4.1 The need for types

Compiling the `Clambda` intermediate code to a typed runtime is not straightforward. First, types are discarded right after type-checking, therefore `Clambda` does not carry types. Second, it is already designed to take advantage of the standard O'Caml runtime environment peculiarities. The standard O'Caml implementation uses a uniform representation to deal with parametric polymorphism. Integer values and pointers toward heap-allocated blocks are both represented by native machine integers and distinguished by a bit of tag. However, when compiling to CIL, integers are typically represented by integers (a value type) and blocks by some reference types. This eventually requires boxing operations on integers in order to make them fit in the same locations as blocks. To achieve that, type reconstruction is required on the `Clambda` code.

The following table shows an example of CIL code generation, which is incorrect because of the involved types are ignored. The variable `t` refers to an array (implemented by an array of objects because of polymorphism):

O'Caml code	
`t.(0) + 1`	
Clambda code	
`(+ (field 0 t) 1)`	
CIL	**Comments**
`ldloc t`	Pushes the local variable `t` on stack.
`ldc.i4.0`	Pushes the integer 0.
`ldelem.ref`	Loads an array element (by reference)
(*)	
`ldc.i4.1`	Pushes the integer 1.
`add`	Computes addition.

At the level of the (*)-marked line, the top of the stack holds a reference to an object whereas the instruction `add` expects an integer value type.

We introduce the `Tlambda` code that carries types and includes type casting operations to address this issue. A type-aware compiler inserts an `unbox` instruction at (*). The type safety property is ensured by the front-end type checking.

4.2 Type re-inference

As sketched in the previous section, retyping `Clambda` allows to compile correct code. Moreover, accurate typing information helps to choose data representations that avoids performance penalties.

4.2.1 Methodology.

We use a retyping algorithm that infers types on the `Clambda` code. In the standard O'Caml runtime, types are all collapsed down to a uniform representation. There is a trade-off: on one hand we need to be as

accurate as possible in order to prevent inefficiencies (typing everything to be an "object" is an option, but a costly one because it maximizes (un)boxing operations), and on the other hand the available information does not allow for much accuracy. We propose the following type grammar:

```
T ::= int | block | string | float
          | closure | unit | any
```

The algorithm propagates type information from the primitives back to the whole code. Having no other clue on source types, there is very little to retype: the types grammar is rather poor, and is based on the types that can be associated with the primitives (handling blocks and integers, but also floats, strings and so on). Distinguishing integers from blocks is a first step. Furthermore, we try to identify particular kinds of blocks wherever possible, in order to manage them specifically. It turns out that some instances of O'Caml blocks: `string`, `float`, `closure` and `unit`, being operated on by specific primitives, can be identified contextually. In order to handle polymorphism, the implementation assigns a representation that inherits from the representation of `block` (which denotes undetermined blocks). The type `any` encompasses every other types. It is mandatory because of parametric polymorphism, and its typical .NET representation is the root class `Object`.

This simple retyping technique only requires a slight adjustement of `Clambda` code to work properly.

4.2.2 *Data representation.*

We translate basic types according to the following correspondences:

O'Caml	bool	int	float	string
CTS	int32	int32	float64	StringBuilder

- We use `StringBuilder`, not `string`, because O'Caml strings are mutable.
- Since types are determined by the way values are used in the intermediate code, O'Caml integers and booleans are mapped to the same representation.

Tuples, arrays, records, lists and sumtype values are traditionally represented by means of heap-allocated, tagged blocks (in the case of a sumtype value, the tag is used to code the involved constructor). These types are not distinguished by the O'Caml runtime and are operated on by the same primitives. Therefore they cannot be identified by type reconstruction. They are all compiled to a common generic representation: arrays of objects (`object[]`), requiring boxing operations on basic type values which are not objects.

Closures are compiled to objects inheriting from `CamIL.Closure`, a dedicated class that declares two methods handling application: `exec` implements total application and `apply: object -> object` is used for partial application. Wrapped around `exec`, `apply` returns a new closure ready to expect the forthcoming arguments, or the final result value, depending on the number of remaining arguments. The closure's environment is stored in object fields.

Mapping an O'Caml class hierarchy to a .NET class hierarchy is very tempting. Besides the theoretical issues it raises (because of the numerous differences between the two object models), this is also hard to achieve because of the internal representation of O'Caml objects: starting from the first intermediate language, objects no longer show up as objects but merely as blocks of fields and functions. O'Caml implements the late binding mechanism by inserting additional code within user code (the standard O'Caml runtime environment was originally designed for the core language, and does not natively support an object layer). The OCamIL compiler processes the corresponding blocks transparently, without knowing they are related to objects.

The current release of the OCamIL compiler was developed according to this design. The back-end approach, using retyping techniques, quickly leads to significant achievements.

4.3 Compatibility

Compatibility is fairly complete. The standard core library, as well as some others (the `graphics`, `threads` and `dynlink` libraries) have been successfully adapted. Large applications have been compiled and behave consistently with the standard implementation.

Let us mention the main differences between OCamIL and the standard implementation. First, some aspects of O'Caml are left implementation-dependent. For example the order of evaluation of function arguments is not specified. The INRIA compiler and OCamIL adopt right-to-left and left-to-right evaluation, respectively. Second, O'Caml provides some partially hidden, low-level and unsafe operations on data representations. OCamIL only emulates a part of them (actually, what is used by the implementation of the standard library). Third, the foreign function interface with C is replaced with a basic interface with CIL methods (more on this topic in subsection 5.1). We focus on managed code until now, but interfacing with unmanaged libraries can be addressed. Finally, the O'Caml data marshaling format is not specified. The OCamIL implementation rely on the BCL serialization API: on one hand, this leads to incompatible data formats and on the other hand, this provides a safe marshaling for free.

4.4 Bootstrapping

We describe here the different steps that lead from OCamIL sources to a bootstrapped compiler running in the .NET framework. Like the O'Caml compiler itself, OCamIL is written in the O'Caml language. More than our personal preferences for O'Caml, it is convenient to use the implementation language of the standard INRIA compiler because we open a new compilation branch on it.

The successive steps needed for building and bootstrapping OCamIL are shown in figure 2. Compiling OCamIL from sources requires the original O'Caml bytecode compiler (`ocamlc`) and runtime (referred to as μ). In the figure, `mlB` stands for the original O'-Caml bytecode.

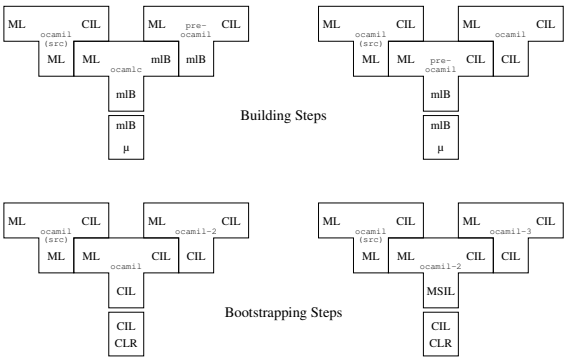

Fig. 2: Building and bootstrapping steps

4.4.1 Building steps

(following figure 2): the hybrid compiler `pre-ocamil` is compiled first. It produces CIL executables and shared libraries from O'Caml source files, but still runs in the standard O'Caml environment. Then we recompile OCamIL sources using the freshly compiled compiler. This produces `ocamil`, which is itself a .NET bytecode executable file. Once this is done, we no longer need the O'Caml system nor the `pre-ocamil` compiler [2].

4.4.2 Bootstrapping steps

(following figure 2): we use the newly built compiler to compile itself. We need two rounds to reach a fixpoint (`ocamil-2` is identical to `ocamil-3`) because of the slight difference of operational semantics exposed in subsection 4.3 (regarding evaluation order). When compiling OCamIL, it affects the ordering of code generation. For that matter, `pre-ocamil` and `ocamil` do not strictly behave the same, so `ocamil` and `ocamil-2` are not strictly identical. In this case it does not affect the semantics of the resulting programs but only their code layout. The additional round fixes the mismatch.

[2] Later on, the `pre-ocamil` compiler should not be used, because it runs in a different world than executables it produces. As explained in subsection 4.3, the O'Caml and OCamIL data marshaling formats are not compatible. This implies that data marshaled by programs compiled by `pre-ocamil` cannot be read back by `pre-ocamil`, a situation that typically happens when compiling from a marshaled abstract syntax tree instead of a source file (as preprocessors generate), or for dynamic linking. This also means that libraries compiled by `pre-ocamil` cannot be used by `ocamil`: they need to be compiled by `ocamil` itself.

4.5 Toplevel Building

The OCamIL compiler and executables compiled by it run in the CLR altogether. Using the .NET dynamic code generation and execution features provided by the reflection API helps building a toplevel utility `ocamiltop`. A toplevel iteratively compiles O'Caml declarations on the fly and executes them, while maintaining a symbol table. Figure 3 displays the toplevel components and shows the processing steps of an O'-Caml expression.

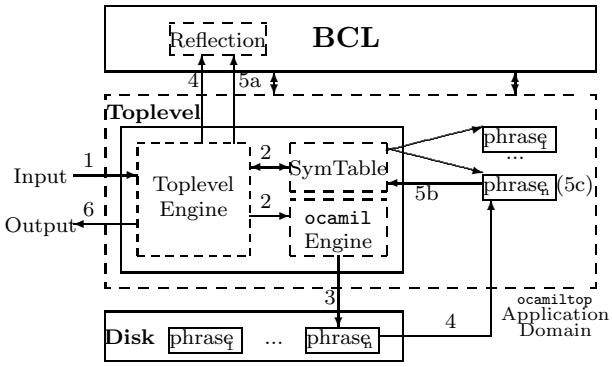

Fig. 3: The toplevel engine

1) The toplevel engine consumes an O'Caml expression $phrase_n$.

2) It uses the `ocamil` compiler engine (together with a Symbol Table resolving free variables) to compile the expression to CIL code.

3) The CIL code is written as a shared library file on the hard disk.

4) The toplevel engine calls the BCL `System.Reflection.Assembly::LoadFrom` method to dynamically load back the emitted assembly to memory.

5a) Calls to the reflection API manage to run a public method of the assembly which was emitted at stage 2. It is a startup method for the compiled expression. 5b) The startup method first registers the bindings defined by $phrase_n$ by accessing directly the table of symbols used by the toplevel. 5c) The startup method then runs the inner code of $phrase_n$ (that may refer to previous expressions using the associations maintained in the table of symbols).

6) The execution flow returns to the toplevel loop that handles output (typically by displaying computed values).

The toplevel prototype writes compiled assemblies to disk, then reloads them back to memory. We plan to develop a new version that directly compiles code to memory: this allows to produce a single assembly that grows up during the toplevel session, from which we expect increased performance.

The toplevel tool is very useful for application development. It also has promising applications using its embedding capabilities.

5. INTEROPERABILITY

OCamIL interoperability capabilities are based on a two-layered approach.

5.1 Basic Foreign Function Interface

The heart of OCamIL interoperability is a simple mechanism which allows to call CIL code from O'Caml programs. It is a replacement of the original O'Caml FFI for C code. OCamIL allows to call static methods written in C# or in bytecode. This was widely used in order to adapt the O'Caml standard library, replacing the C code by calls to the .NET BCL. However, this is limited and not type-safe: its main purpose is to support safe, high-level communication.

5.2 O'JACARE.NET

We provide a high-level, safe interfacing of O'Caml and C# through their object models, using an IDL approach. We have developed a tool called `O'Jacaré.net` that compiles IDL files and generates all necessary wrappers to mix components written in both languages. Details can be found in [10].

5.2.1 *Comparing object models.*

Type systems and object models can be interleaved in many ways. There are important differences between the object models of O'Caml and C#. For instance, class declarations allow multiple inheritance and parametric classes in O'Caml but not in C#, method overloading and class downcasting are only supported in C# (but in O'Caml the type of `self` can appear in the type of a method eventually overridden in a subclass). The intersection of the two models corresponds to a simple class-based language, for which inheritance and subtyping relations are equivalent, overloading and binary methods are not allowed. For the sake of simplicity, it does not offer multiple inheritance nor parametric classes. This model inspires a basic IDL for interfacing C# and O'Caml classes.

5.2.2 *Encapsulation.*

In contrast to direct external calls presented above, using `O'Jacaré.net` is safe and much more expressive. O'Caml programs can allocate C# objects and call instance methods. It is also possible to inherit C# classes in O'Caml and redefine methods. Late-binding is transparently performed between the two languages. The other way around is also possible: libraries compiled by OCamIL can expose classes that will be used in C# programs.

This requires a tricky implementation because O'Caml objects are no longer objects at run-time. The mechanism that enables late-binding to run back and forth between O'Caml and C# worlds is illustrated in figure 4. In this example, a C# component defines the well-known didactical classes `Point` and `ColoredPoint` that are exposed in an IDL file.

The compilation of this file generates the corresponding O'Caml wrappers, allowing to allocate objects and call methods upon the foreign C# classes as if they were native. New O'Caml classes, such as `colored_point_ml` in the figure, can inherit from them.

However, a complete and proper cross-language late-binding mechanism cannot be implemented with such a simple design. Let us assume that `Point` defines a method `toString`, and that `ColoredPoint` both defines a method `getColor` and overrides the definition of `toString` by concatenating the results of a call to the method `getColor` and a call to the method `toString` of the superclass. If we redefine the `getColor` method in O'Caml, and expect the `toString` method to be specialized through late binding, we need to produce an additional stub in each language: a call to `toString` on `colored_point_ml` traces back to the `ColoredPoint` class, which has no idea of the O'Caml instance and thus of the redefinition of `getColor`.

The two stubs hold a reference to each other. The C# stub, named `ColoredPointStub`, overrides each method as a callback to the O'Caml stub `callback_colored_point` and the latter defines each method as a non-virtual call to `ColoredPoint`, the base-class of the former. Following figure 4, the O'Caml class `mixed_colored_point` inherits from the O'Caml stub class. Thanks to the non-virtual call, a call to the `toString` method traces back to the implementation of `ColoredPoint` Then the virtual call to `getColor` is late bound to `ColoredPointStub`, which virtually calls the O'Caml corresponding method on `callback_colored_point`, falling back on O'Caml late-binding mechanism.

5.2.3 *Blending two object models.*

`O'Jacaré.net` allows to partially handle both object models. [10] gives examples of C# objects downcasting and multiple inheritance of C# classes in O'Caml.

We need the IDL glue to interoperate between O'Caml and C#: because of design and semantics differences, encapsulation is needed in both ways. However, we benefit from sharing the same runtime envi-

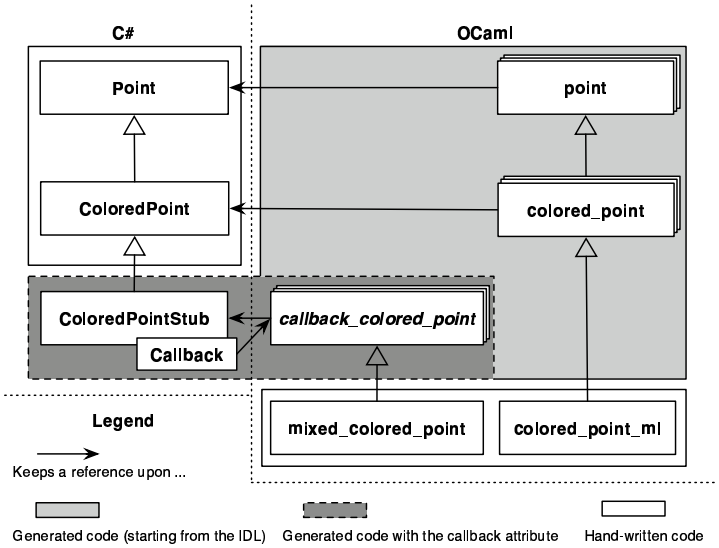

Fig. 4: Relationship between classes

ronment. The communication between components is type safe and we take advantage of unified garbage collection and thread management.

6. APPLICATIONS

Adapting O'Caml to .NET is interesting for both communities. We believe it can help make popular O'Caml applications, and that new possibilities are offered by interoperability. Let us mention a few of them.

O'Caml is given access to new libraries. O'Caml programs can use libraries ranging from graphical toolkits to remoting facilities. They can be distributed as applets that run inside a browser's windows. See figure 5 for an example of O'Caml applet, that runs a raytracer (the winning entry of the ICFP 2000 programming contest). Using `O'Jacaré.net`, the same O'Caml program can be given a graphical user interface written in C#.

See also figure 6 for an O'Caml toplevel embedded in a graphical interface written in C#.

O'Caml benefits from new tools. We can already use .NET tools such as debuggers or profilers on OCamIL programs. It is also possible to integrate the O'Caml language in IDE such as Visual Studio.NET.

.NET is enriched by O'Caml. It is important to promote programming paradigms such as functional programming. Moreover, the O'Caml object layer can interest OO programmers and encourage them to give O'Caml a try. O'Caml is particularly good at tree manipulations or symbolic computations, some of the fields where languages such as C# cannot stand the comparison. Syntactical tools such as Camlp4 [11], which was successfully compiled by OCamIL, can open new tracks for writing compilers, using O'Caml as a target language. The possibility to embed an O'Caml toplevel component in C# applications also offers interesting perspectives.

7. RELATED WORK

The approach described for `O'Jacaré.net` (two runtime environments running side by side) has also been used in other projects.

The Haskell interpreter, Hugs98 for .NET [12], allows .NET classes. Its implementation is based on a mechanism similar to O'Caml / `O'Jacaré.net`. At the level of source language, it allows a basic communication with the .NET platform which allows thorough communication to be built upon and used through a high level language construction. Automatic code generation with a dedicated tool is needed to achieve it. As for execution, it provides two virtual machines (interpreter and CLR) running simultaneously. The Dot-Scheme [13] project implements a FFI (Foreign Function Interface) to the .NET platform from PLT Scheme. Here again, execution is performed by two virtual machines. At the language level, the implementation (based on CLR introspection capabilities) allows an easy and direct .NET classes manipulation.

The current trend is to directly produce bytecode for either Java (cf MLj [14]), or .NET. For .NET, a lot of works have been done :

- for SML: SML.NET [15] and MoscowML for .NET [16];
- for O'Caml: F# [17] and OCamIL.

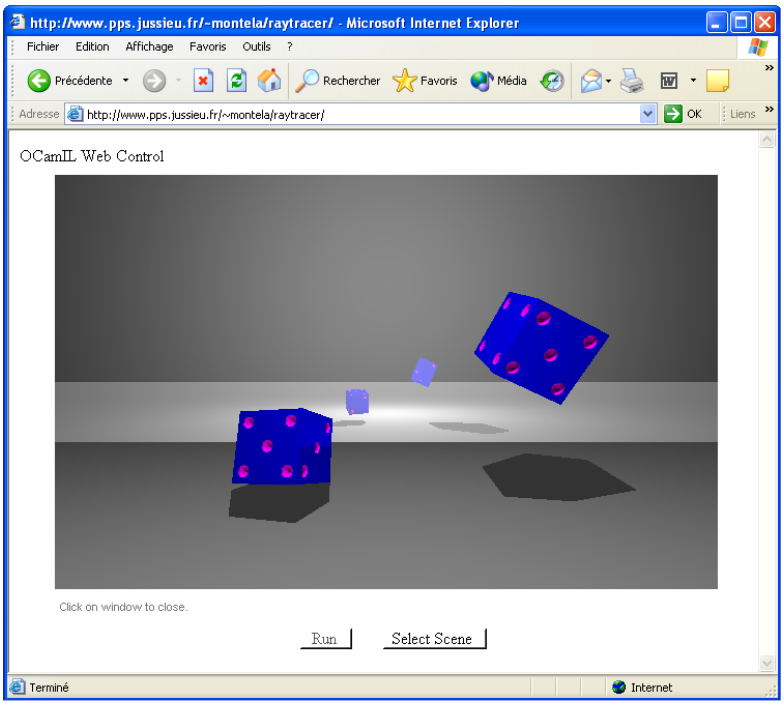

Fig. 5: An applet running a raytracer written in O'Caml.

The main interest to use the same runtime is to facilitate memory management (GC) and multi-threading.

F# and OCamIL illustrate two different views of interoperability. F# conception is focused on interoperability. Its purpose is to manipulate the .NET proposed object model in a functional / imperative language similar to CamlLight. The outcome is a new Caml dialect using .NET object model. But the .NET object model is really far from the O'Caml object model. The advantage is to directly manipulate CTS types, with no additional tool and in a natural way. It provides a comfortable programming and allows an implementation as direct as possible (which guaranties better performance).

On the other hand, the used object model is not integrated as well in the functional paradigm as the O'Caml model. In many cases, it is mandatory to help the type inference by giving types annotations for CTS. Then, parametric polymorphism and row polymorphism become a kind of interfaces polymorphism when .NET methods are called.

On the contrary, OCamIL does not modify the original language. There are no new constructs coming from the target architecture and the interoperability is managed accross the O'Caml object model.

There are two main consequences :

- the difference between the two object models forbids a direct compilation from O'Caml objects to the CTS;
- this inadequacy makes it necessary to generate stub classes (we compile IDL files with our tool `O'Jacaré.net`).

To put it shortly, F# is for the C# programmer who wants to use functional programming, and OCamIL is for the O'Caml programmer, who wants to take advantage of the .NET environment without changing his favorite language.

MLj and SML.NET join together the two approaches by proposing the essence of SML on the Java and .NET platforms, and integrating the C# object model (but it is true that without object features in the original languages there is no decision to select an object model). MoscowML for .NET only allows static method calls.

From the Scheme side, the Bigloo compiler allows to compile to the JVM or the CLR runtimes. As for DotScheme, the .NET features are nicely incorporated in the Scheme language by using special functions and macros. The Scheme language fits well in an interoperability setting: its syntax is easily extensible and its dynamic typing facilitates the integration of new features. Dynamic typing is more in the spirit of the Java and .NET platforms that propose many services of instrospection.

Although Eiffel is not a functional language, its .NET version [18] encounters similar difficulties than OCamIL. The two object models have a multiple inheri-

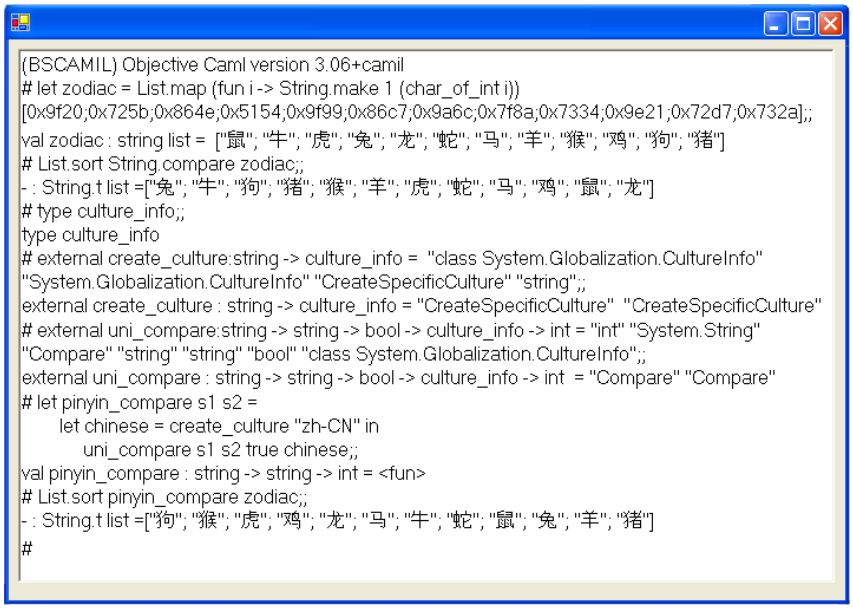

Fig. 6: A toplevel session in a C# window, demonstrating culture-specific ordering.

tance, parametric classes and no overloading. However their techniques of compilation strongly differ. Eiffel relies on CTS interfaces to emulate multiple inheritance.

8. RETYPING TECHNIQUES AND FUTURE WORK

For the sake of compatibility and front-end independence, OCamIL currently adopts a back-end approach that leads to retype an intermediate language from scratch. We are currently developing an alternative implementation which retrieves source types from the O'Caml type-checking step. Let us compare the pros and cons of each technique.

8.1 What hinders the strict back-end approach

As mentioned in subsection 4.2, the retyping technique requires the front-end to be slightly modified. The heart of the problem are data types with non-uniform representations such as sumtypes. Here is a sample sumtype definition:

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

The sumtype `t` declares two constant constructors and a non-constant constructor. As for the O'Caml runtime, these are respectively represented by integers `0`, `1` and a pointer to a block containing another value of type `t`. This is homogeneous in the O'Caml runtime but the retyping algorithm eventually infers two different types, `int` and `block`, for values of type `t`. Consider the following function and its compiled representation in `Clambda` code:

O'Caml code	`Clambda` code

`let cut x =`
`  match x with`
`  | Node n -> n`
`  | x -> x`
	`let cut = closure(cut):`
	`  x ->`
	`  if (isint x) then x`
	`  else (field 0 x)`

Type	Inferred type
`t -> t`	`Sumtype -> Sumtype`

The `isint` primitive tests the bit of tag that distinguishes integers from pointers on blocks. In order to take the duplicity of the parameter `x` into account, the grammar of reconstructed types needs a new item `Sumtype`, that represents the union of `int` and `block`. The function `cut` above receives the type `Sumtype -> Sumtype`. We do not want to use the general-purpose type `any` here to give a chance to `Sumtype` values to be mapped to a more precise and adequate type than `Object`. Of course, applying `cut` to constants requires boxing operations. There is something wrong though, as the following example reveals:

O'Caml code	`Clambda` code

`let hell a b =`
`  match a with`
`  | Zero`
`  -> One`
`  | _ -> b`
	`let hell = closure(hell):`
	`  a -> b ->`
	`  if (isint a) then`
	`  (if (a != 0) then b`
	`  else 1)`
	`  else b`

Type	Inferred type
`t -> t -> t`	`Sumtype -> int -> int`

The type of the parameter `b` is problematic. Looking at the O'Caml source code we know that `a` and `b` are both of type `t`. But looking at the `Clambda` code, one is tempted to claim that `b` is an integer! The only in-

formation that the re-typing algorithm has about `b` is that its type is unifiable with `int` (because of the subexpression: `if (a != 0) then b else 1`). Following the policy of being as accurate as possible, `b` is typed to be an integer, and the function `hell` receives the type `Sumtype -> int -> int`. Later on, when compiling an application such as `hell One (Node Zero)`, the retyping algorithm detects inconsistency and aborts. In general, the algorithm cannot backtrack and give `b` a correct type: the definition of `hell` and its applications can reside in separately compiled modules.

Fortunately, there is a simple workaround. Changing the representation of sumtypes a little bit is a quick modification of the compiler. Because constant constructors can be encoded as empty blocks (the tag of the block coding the constructor), we uniformly represent sumtypes by blocks [3]. This avoids the multiplicity of representations for the same type that caused types reconstruction errors. Although this is achieved by a slight modification of the compiler, this somehow betrays the spirit of the back-end approach.

8.2 Types propagation

The retyping of the `Clambda` intermediate language is not accurate enough, entailing costly data structure allocation (object arrays). Data access is slowed down by dynamic typechecking and boxing operations. Retrieving exact types allows to compile data to adequate representations: for instance each constructor of a given sumtype can be implemented as an object with fields holding the parameters of constructor, with their exact types. We propose to modify the implementation of O'Caml in order to propagate typing information along intermediate languages from the type-checking step until the `Clambda` code. Maintaining OCamIL up to date with the latest O'Caml release will be harder because types are likely to evolve along with O'Caml development, but as explained in the previous subsection a strict back-end implementation quickly reaches its limits anyway. Future work will focus on implementing and exploiting type propagation, and we expect important performance improvements. Type propagation also has applications in debugging O'Caml programs, because the generated CIL will have more adequate types with respect to the O'Caml source program.

9. CONCLUSION

Java's success has popularized bytecode-based runtimesthat offer modern techniques to improve safety, such as typed bytecode, garbage collection and built-in security policies. The .NET CLR is based on a similar design, and tries to improve security. These two platforms help portability, interoperability and offer a convenient target for compiler implementors.

The OCamIL project helps to evaluate the .NET platform and the O'Caml implementation with respect to each other. The .NET CLR is presented as a runtime of choice to run multi-languages applications, which implies a stricter control over pieces of code and the addition of new features to the execution platform, in order to support more programming features. However, these efforts have been mainly object-oriented: originally for C#, Visual Basic and C++. Logical and functional paradigms are not natively supported. Closures and advanced flow-control (even exceptions) implementation is too costly. Likewise, parametric polymorphism does not fit well in the object models of today's runtimes. Fortunately, there are promising developments towards these directions (such as ILX and generics).

Symmetrically, language implementations need to adapt to new runtimes. Compiling to a typed virtual machine raises new issues that were not relevant in dedicated functional virtual machines [19]: now type information is needed down to bytecode generation. To address efficiency issues, types have to be as accurate as possible, ideally by propagating the static type-checking step information. Appel's slogan "Runtime Tags Aren't Necessary" [20] does not hold anymore.

For the sake of compatibility and front-end independence, OCamIL has adopted a back-end approach that leads to retyping an intermediate language from scratch. We are currently developing an alternative implementation which retrieves source types from the O'Caml type-checking step. The solution needs to modify the implementation of O'Caml in order to propagate typing information along intermediate languages from the type-checking step until the `Clambda` code, which is successfully experimented with the development version of OCamIL.

Despite these inadequacies, the .NET platform has proven to be an interesting framework to develop a compiler for. The OCamIL compiler and toplevel allow the development of O'Caml applications for the .NET platform, with the guarantee of compatibility with O'Caml (including advanced programming features) and managed CIL code production. Other .NET languages can consume O'Caml components, for instance the OCamIL toplevel can be embedded inside a C# application, to produce dynamically compiled O'Caml code.

[3] A more complex policy can be imagined for sumtypes: represented by integers if made of constant constructors only, and represented by blocks otherwise. However this is not appropriate for O'Caml polymorphic variants which can be incrementally extended, for example by adding a non constant constructor to a set of constant constructors.

10. REFERENCES

[1] Thai, T.L., Lam, H.: .NET Framework Essentials. 3rd edn. O'Reilly Edt (2003)

[2] Stutz, D., Neward, T., Shilling, G.: Shared Source CLI. O'Reilly Edt (2003)

[3] Dumbill, E., Bornstein, N.: Mono: A Developer's Notebook. Developers' Notebooks. O'Reilly Edt (2004)

[4] Leroy, X.: The Objective Caml system release 3.06 : Documentation and user's manual. Technical report, Inria (2002) `http://caml.inria.fr`.

[5] Montelatici, R., Chailloux, E., Pagano, B.: OCamIL homepage (2004) `http://www.pps.jussieu.fr/~montela/ocamil`.

[6] Syme, D.: ILX: Extending the .NET common IL for functional language interoperability. Electronic Notes in Theoretical Computer Science **59** (2001)

[7] Kennedy, A., Syme, D.: Design and Implementation of Generics for the .NET Common Language Runtime. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), ACM SIGPLAN (2001)

[8] Yu, D., Kennedy, A., Syme, D.: Formalization of Generics for the .NET Common Language Runtime. In: Proceedings of the 31st Symposium on Principles of Programming Languages (POPL), ACM SIGPLAN (2004)

[9] Remy, D., Vouillon, J.: Objective ML: An effective object-oriented extension to ML. Theory and Practice of Object Systems **4** (1998) 27–50

[10] Chailloux, E., Henry, G., Montelatici, R.: Mixing the Objective Caml and C# programming models in the .NET framework. In: Proceedings of Int. Worshop on Multiparadigm Programming with OO languages (MPOOL'04). (2004)

[11] de Rauglaudre, D.: camlp4 : Reference manual. Technical report, Inria (2002) `http://caml.inria.fr`.

[12] Finne, S.: Hugs98 for .NET homepage (2003) `galois.com/~sof/hugs98.net`.

[13] Pinto, P.: Dot-Scheme: A PLT Scheme FFI for the .NET framework. In Flatt, M., ed.: Scheme Workshop. (2003) 16–23

[14] Benton, N., Kennedy, A.: Interlanguage Working Without Tears: Blending SML with Java. In: International Conference on Functional Programming. (1999)

[15] Benton, N., Kennedy, A., Russo, C., Russell, G.: sml.net homepage (2005) `www.cl.cam.ac.uk/Research/TSG/SMLNET/`.

[16] Kokholm, N., Sestoft, P.: Moscow ML .Net Internals. (2003) `http://www.dina.dk/~sestoft/mosml.html#mosmlnet`.

[17] Syme, D.: F# homepage (2005) `http://research.microsoft.com/projects/ilx/fsharp.aspx`.

[18] Simon, R., Stapf, E., Meyer, B.: Full eiffel on the .net framework. MSDN Library, `http://msdn.microsoft.com/library/default.asp?url=/library/en-uspdc_eiffel.asp` (2002)

[19] Leroy, X.: The effectiveness of type-based unboxing. In: Workshop on Types in Compilation. (1997)

[20] Appel, A.: Runtime tags aren't necessary. Lisp and Symbolic Computation (1989)