Objective Caml on .NET: The OCamIL Compiler and Toplevel

Raphaël Montelatici[1], Emmanuel Chailloux[2], and Bruno Pagano[3]

[1] Equipe Preuves, Programmes et Systèmes (CNRS UMR 7126)
Université Denis Diderot (Paris 7) - 2 place Jussieu, 75005 Paris, France
Raphael.Montelatici@pps.jussieu.fr,
[2] 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,
[3] Esterel technologies, 679 Av Julien Lefèbvre
06270, Villeneuve-Loubet, France
Bruno.Pagano@esterel-technologies.com

Abstract. We present the OCamIL compiler for Objective Caml that targets .NET. Our goal is to understand whether this new generation of virtual machines and runtime environment can help us compile ML programs and produce executables of reasonable efficiency. We aim at compatibility with the original language, and its advanced programming features (functional values, exceptions, parameterized modules, objects). We detail the bootstrapping cycle producing OCamIL itself as a .NET component. This entails the building of an interactive loop (toplevel) which may be embedded within .NET applications.

1 Introduction

The .NET[4] 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 compiler generates portable MSIL bytecode (*MicroSoft Intermediate Language*). By assuming compliance to the CTS type system, components interoperate safely.

This has motivated the adaptation of numerous languages, such as C#, J#, A#[5] Eiffel[6], Scheme[7], Sml[8], F#[9], P#[10], Mercury[11].

[4] www.microsoft.com/net
[5] www.usafa.af.mil/dfcs/bios/mcc_html/a_sharp.html
[6] www.msdnaa.net/Resources/display.aspx?ResID=811
[7] www-sop.inria.fr/mimosa/fp/Bigloo
[8] www.cl.cam.ac.uk/Research/TSG/SMLNET/
[9] research.microsoft.com/projects/ilx/fsharp.aspx
[10] www.dcs.ed.ac.uk/home/jjc/
[11] www.cs.mu.oz.au/research/mercury/dotnet.html

Even though the main implementation runs under Windows, some Open Source efforts adapt .NET for Unix BSD and Windows (Rotor[12]) and Linux (Mono[13]). That recalls Java's motto : "COMPILE ONCE, RUN EVERYWHERE".

We eventually get a safe and efficient multi-language platform with a unique runtime, that could run on different systems. We intend to check this claim by writing a .NETcompiler for our beloved language, Objective Caml[1].

Objective 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 byte-code and a native code compiler, which generates efficient programs. However, new virtual machines like Java's JVM or .NET CLR are not necessarily relevant for functional languages, because functional values do not fit well in an object model. Moreover, static typing is ignored by the corresponding runtimes, which perform typechecking at code loading time. Appel's motto "Runtime Tags Aren't Necessary" [2] does not hold anymore.

The OCamIL[14] compiler can help widespread Objective Caml applications. Hence, its main constraint is compatibility with Objective Caml. To achieve that, we open within the Objective Caml compiler itself a new code generation branch that generates typed MSIL. The OCamIL compiler is written in Caml itself, enabling bootstrapping as a severe compatibility test. Taking advantage of the .NET reflection API, OCamIL can dynamically execute the code that it produces, a feature that lead us to build a toplevel interaction loop. Both compiler and toplevel can be redistributed as .NET components.

We first present the relevant features of .NET platform from a compiler writer's point of view, then describe OCamIL implementation and detail the steps leading to a bootstrapped compiler, and a toplevel system. The toplevel runs as a .NET component and therefore can be embedded in any .NET application, adding the power of Objective Caml machinery to third-party components. We also present application examples and finally test OCamIL against other ML compiler, such as F# and SML.NET. A short conclusion outlines our future work.

2 .NET Platform

Microsoft claims the .NET platform to be the next reference technology in the development of desktop applications as well as smart clients and Web services. It is supposed to enhance security and error management, and should help getting rid of Windows shared libraries issues [15].
The .NET platform specifies a runtime environment[16] mainly composed of a

[12] `msdn.microsoft.com/net/sscli`
[13] `www.go-mono.com/`
[14] `www.pps.jussieu.fr/~montela/ocamil`
[15] "The End of DLL Hell", `msdn.microsoft.com/netframework/`
[16] CLR : Common Language Runtime.

stack-based virtual machine[17] and a support library (BCL[18]). The virtual machine runs a so-called MSIL bytecode and checks whether it is compliant with respect to a typed class model CTS. The files which package executable MSIL opcodes together with eventual inlined ressource data are called PE files[19]. At runtime, the bytecode is Just-In-Time compiled to machine code, as sketched in figure 1.

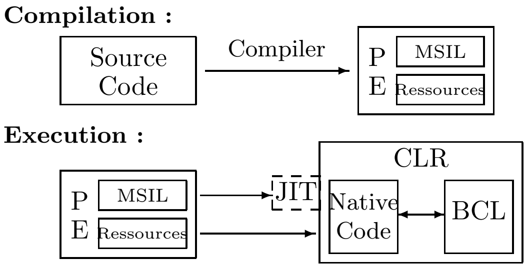

Fig. 1. Compilation and execution

Let us detail the main features of the .NET platform :

MSIL bytecode It looks like typed machine code : each location where values are stored or passed across, is given a type. We distinguish Reference Types (for heap-allocated objects) from stack-allocated Value Types (which are not reduced to base types and may have a complex structure). Bytecode instructions (`box` and `unbox`) can be used to switch between the two kinds of representation. Objects have fields and methods which may use a pool of local variables. As for control, the virtual machine handles method calls (with an optional late-binding mechanism or even through indirection), branching within a same method and exceptions.

Deployment The base .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 also helps versionning and localization management.

Execution Safety An executable file can be made of components compiled from different source languages. In order to avoid typing inconsistencies, the

[17] VES : Virtual Execution System.
[18] Base Class Library.
[19] Portable Executable files

MSIL code has to be statically typed. Moreover, any assembly compiled for the CLR can be checked by a verification tool `PEVerify` that detects stack inconsistencies, errors while resolving external assemblies dependencies (for instance erroneous calls to externally declared methods), and even some runtime typing errors. Typing information is kept along with code and is used at runtime : in case of a dynamic type error, the CLR raises an exception. These features are greatly valuable for the development of a compiler that targets MSIL. The MSIL bytecode conforming to typing and verification constraints is called “managed code”. However the platform enables calls to unmanaged code, which is still necessary for low-level operations.

Note that the runtime environment features a Garbage Collection mechanism, which frees the developper from tricky memory management issues.

Performances The platform relies on a systematic Just In Time compilation mechanism (there is no complicated heuristic here : each method is natively compiled at first call). It is possible to bypass this behaviour by pre-compiling an assembly to a native image.

Methods can be tail-called (*i.e.* without stacking a new method frame), which is particularly useful for functionnal languages implementation.

Reflection Last but not least, the platform features a fairly complete reflection library, which enables dynamic code management (generation, loading and execution).

3 The OCamIL Compiler

3.1 General Scheme

Our main goal is to port Objective Caml to the .NET platform and be as compatible as possible with the reference implementation. Efficiency issues are left aside in the first place. Writing a new compiler from scratch, for a modern functional language like Objective Caml, with imperative and object-oriented features, a parametric module system and a static typing system with type inference, is not an easy task. Our experiment consists in writing a compiler that takes advantage of the standard INRIA compiler by modifying its back-end component. We branch on the standard Objective Caml compilation chain, after parsing and typing operations. We do not compile a source file from scratch : we get the internal representation `Clambda` from the standard Caml compiler, see figure 2 for details. `Clambda` explicitly manages closures.We introduce a new intermediate representation `Tlambda` which rebuilds types information : we discuss its use in sub-section 3.4.

OCamIL compiles a `.ml` Caml implementation file to a `.cmx` object file (a specific file format which mainly is roughly MSIL with unresolved references), and links a list of `cmx` files to a single assembly in a portable executable file, that references external assemblies such as the BCL components and the OCamIL

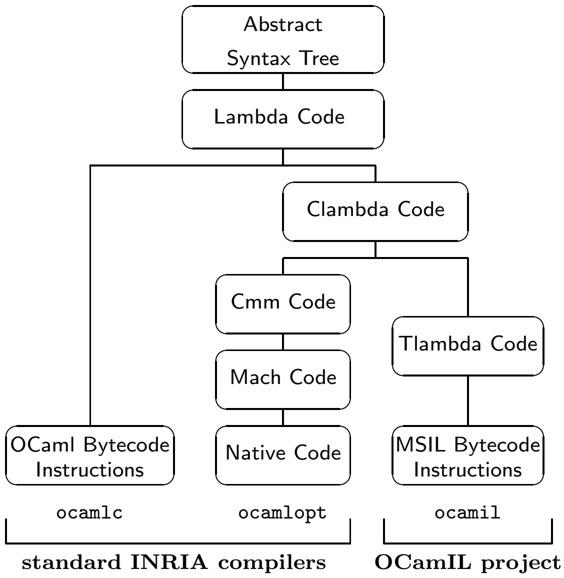

Fig. 2. OCamIL inside Objective Caml.

runtime support library `core_camil.dll`. It may also produce shared library `.dll` files. The generated code relies on the CLR environment and its libraries in order to perform low-level tasks (memory management, IO . . .).

3.2 Data Representation

Basic Types Mapping Caml base types to .NET types is not a big deal, we translate base types according to the following correspondences :

Caml	bool	int	float	char	string	unit
.NET	int32	int32	float64	char	StringBuilder	void

- Objective Caml strings are mutable, so we cannot use the base type `string`.
- .NET has unicode built in, so we have 16-bits characters. A strictly-compliant version of OCamIL library restores original 8-bits Caml characters.

Structured Values Tuples, arrays, records, lists and sum type values are traditionnaly represented in memory by means of heap-allocated tagged blocks (in the case of a sum type value, the tag is used to code the involved type constructor).

We compile such blocks to object arrays (`object[]`), which forces us to box base type values which are not objects.

Closures Caml closures are represented as a pair of a function pointer and an environment, they are special instances of heap-allocated blocks. They can represent mutually recursive functions (by means of sharing and cycling constructs).

Caml Objects Mapping an Objective 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 objects in the Caml intermediate code : objects do not show up anymore as objects but only as blocks of fields and functions, the late binding mechanism being explicitely added to the program code (this is because the standard Objective Caml runtime environment was originally designed for Caml core language, and does not natively support the object layer).

3.3 Execution Control

Application The OCamIL implementation compiles closures to classes inheriting the dedicated `CamIL.Closure` class. It has fields used to store the closure's environment and two main methods : an `exec` method implementing the function itself (in case of total application) and an `apply: object -> object` method used in case of partial application, which returns the new closure resulting of the application of the next expected argument.

Exceptions The Caml exceptions are directly implemented by using the exception mechanism of the target platform, by means of a `CamIL.Exception` class which inherits `System.Exception` (the root class of all exceptions in CTS).

3.4 Intermediate Language Typechecking

The OCamIL compiler gets a preprocessed representation of a Caml program : namely `Clambda` intermediate code. This code dramatically lacks types information, and what is even worse, it is already designed to take advantage of the standard Caml runtime environment peculiarities. For instance, the standard Caml implementation processes integer values and pointers toward heap-allocated blocks the same way, distinguishing them by means of a tag bit. Hence allocating a Caml block which contains an integer does not require any indirection, since the integer can be inlined inside the block. This contrasts with MSIL block representation (using objects arrays) which requires allocation of boxed representation of the integer.
This kind of manipulation produces more complex code, and requires an analysis of the `Clambda` code which aims at rebuilding a partial type information. The following table shows a case of MSIL code generation, which is incorrect because it does not know about the types involved (the variable `t` refers to an array) :

Source Caml	MSIL	Comments
`t.(0) + 1`	`ldloc t`	Local variable `t` pushed on stack.
	`ldc.i4.0`	Integer 0 pushed on stack.
Clambda code	`ldelem.ref`	Loading of array element (by reference)
`(+ (get t 0) 1)`	(*)	
	`ldc.i4.1`	Integer 1 pushed on stack.
	`add`	Addition.

At the level of the (*)-marked line, the top of the stack stores a reference to an object whereas the addition instruction `add` expects an integer value type. We need a new intermediate language `Tlambda`, which carries types and introduces type casting operations. The previous code generation by a type-aware compiler would have inserted an `unbox` instruction at (*). Type safety is ensured by the property of the original typing by Caml during the first steps of the compilation chain.

3.5 Foreign Function Interface

Even though interoperability issues were not our initial matter, we did implement a simple interoperability mechanism which allows to directly call MSIL code from Caml programs. Caml handles external calls to library functions written in C. We had to provide a similar mechanism for MSIL-compiled programs. This has been widely used in order to adapt the Objective Caml standard library to OCamIL. C calls have been replaced by calls to static methods written in C# or in bytecode, that take advantage of the .NET library. Some calls do not even need additional stub code on .NET-side, as in the following example taken from the module `Sys` of standard library.

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

let getenv var =
  let s = il_getenv var in
    if s = "" then raise Not_found else s
```

The OCamIL version is made of a touch of Caml code wrapping a direct call to the BCL `GetEnvironmentVariable` static method.

Note that the foreign function interface is still low-level and is not type safe.

3.6 OCamIL building and bootstrapping

We describe here the different steps that lead from the working sources of OCamIL to its executable form as a .NET assembly.

Like the Objective Caml compiler itself, OCamIL is written in the Caml language. In addition to our personal liking for Caml to write a compiler, it is actually natural to use the same implementation language as the standard INRIA compiler since we open a new compilation branch on it.

Compiling OCamIL from sources supposes one has got a working installation of Objective Caml bytecode compiler and linker (which implies having the Objective Caml execution engine as well).

The first steps of OCamIL build are shown in figure 3. We explicitely distinguish Caml et OCamIL compilers and linkers, refered to as `ocamlc-c`, `ocamlc`, `ocamil-c` and `ocamil`. On the figure, `.cmo` refers to standard Caml object files and `mlB` stands for the original Caml bytecode.

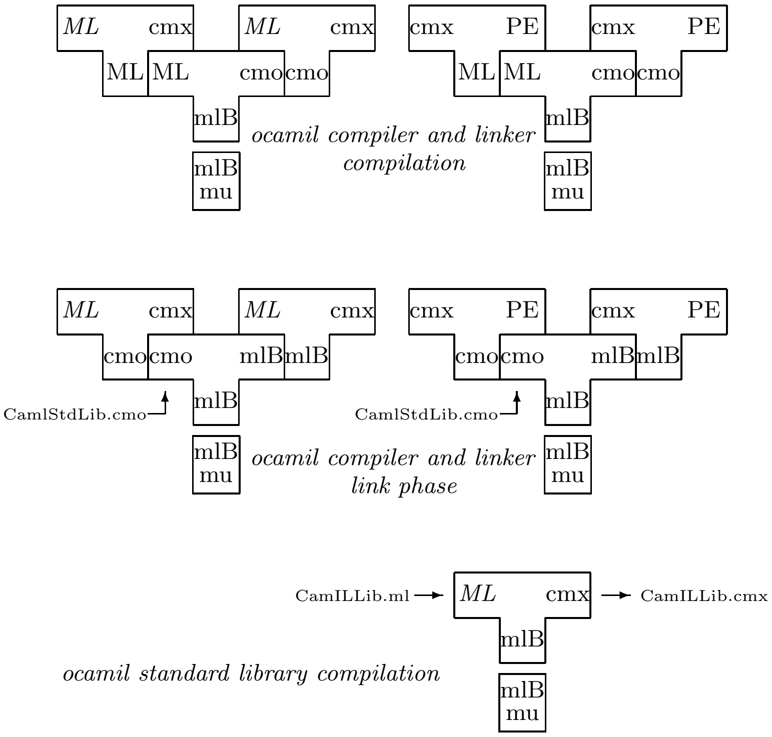

Fig. 3. Building OCamIL : first round.

Following figure 3, we get :

- the OCamIL compiler and linker, which produce MSIL portable executables and shared libraries from Caml source files [20].
- Objective Caml libraries compiled for .NET using the latter compiler. They had to be partially rewritten for OCamIL, replacing external C functions

[20] The source language is actually slightly different from Objective Caml, since it does not embed external C calls anymore, but MSIL calls : that is why we write *ML* instead of ML.

calls to external MSIL methods calls. As to the major part of Caml, which is written in the Caml language, the high compatibility of OCamIL with the standard compiler allows us to the code unchanged.

This composes a working toolkit to compile Objective Caml programs to .NET platform. However, it is an hybrid system, because it produces MSIL bytecode while itself a regular Objective Caml bytecode executable, which requires the Caml bytecode machine.

Having adapted the main part of the Objective Caml standard library, it becomes possible to compile OCamIL sources using OCamIL itself, as depicted in figure 4.

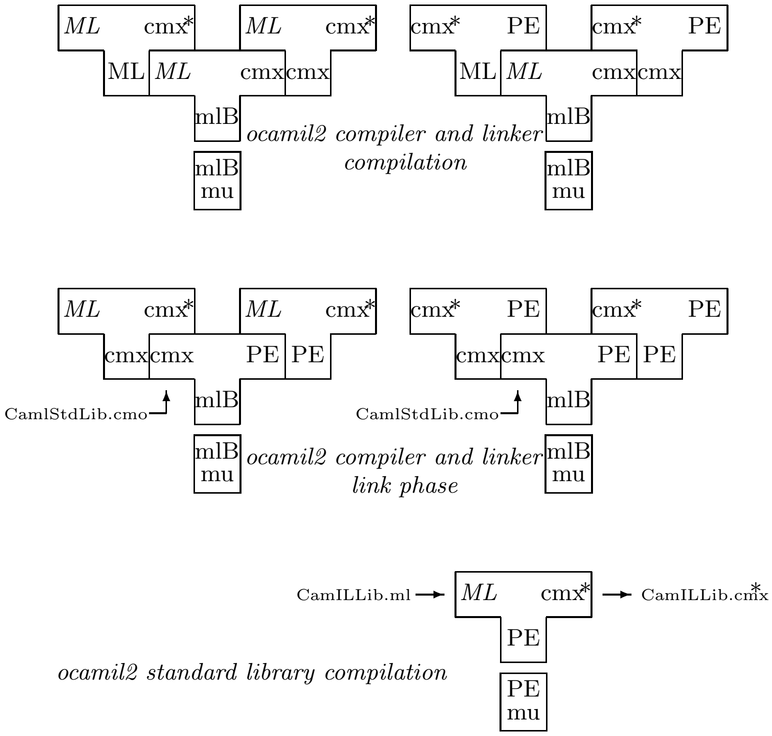

Fig. 4. Building OCamIL : second round.

We first compile the OCamIL source files using the Objective Caml bytecode executable version of OCamIL and get an MSIL implementation of it, `ocamil2`. However, `ocamil` and `ocamil2` are not fully compatible, because we replaced data marshalling C primitives as rough calls to the BCL serialization API (located in `System.Runtime.Serialization` namespace), leading to an inconsistent data representation. Thus, `ocamil2` is not able to deserialize values marshalled by `ocamil`, typically library object files, which is what we intend to point out by distinguishing the `.cmx` and `.cmx*` file formats on figure 4.

In order to get a fully working compiler, we have to recompile the standard library files using the new compiler. This finally leads to a OCamIL executable running under the .NET environment, which can be redistributed as any .NET application, and does not need the standard Objective Caml compiler and runtime system anymore.

We can give it a serious test with an additional bootstrapping cycle, that is recompile OCamIL using `ocamil2`.

3.7 Toplevel Building

From now on the OCamIL compiler runs in the same world as the executables it produces. By using the .NET dynamic code generation and execution features, provided by the reflection API, we can build a toplevel utility `ocamiltop`. A toplevel repeatedly compiles Objective Caml declarations on the fly and executes them, while maintaining a symbol table. Figure 5 displays the toplevel components and the way they operate to compile a Caml expression.

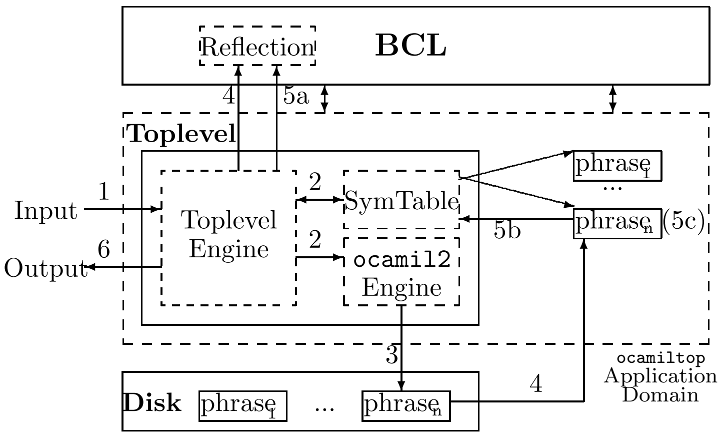

Fig. 5. Toplevel engine

1. The toplevel engine consumes an Objective Caml expression $phrase_n$.
2. It uses `ocamil2` compiler engine (together with a Symbol Table resolving free variables) to compile the expression to MSIL code.
3. The MSIL code is written as a shared library file on the hard drive.
4. The toplevel engine then calls `System.Reflection.Assembly::LoadFrom` to dynamically load the corresponding assembly within its application domain (in memory).

5. a) A call to `System.Reflection.Assembly::GetType` gives access to a predefined class that defines a public startup method, which is immediately runned using a call to `System.Type::InvokeMember`. b) The startup method first registers the bindings defined by $phrase_n$ by accessing directly the table of symbols used by the toplevel. c) The startup method then runs the inner code of $phrase_n$ (that may refer to previous phrases using the associations maintained in the table of symbols).
6. Execution flow returns to the toplevel loop, which handles output (typically by displaying computed values).

Our prototype generates compiled assemblies to disk, then reloads them to memory. A future version will compile directly to memory : it is more effficient and allows to produce a unique assembly that grow up during the toplevel session.
The toplevel utility is very handy for application development. It also has promising applications using its embedding capabilities, see sub-section 4.1.

4 Applications and Tests

4.1 Embedded Toplevel

Objective Caml programs compiled to .NET platform may export functions to be run from other pieces of software. Our work allows to embed a Objective Caml toplevel (`ocamiltop`) inside other applications. For instance, we easily developped a graphical interface in C# for `ocamiltop`, see figure 6.

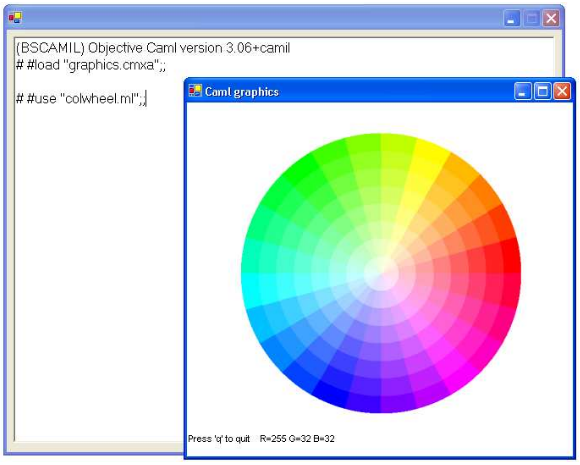

Fig. 6. Graphics example in embedded toplevel.

The figure shows a simple toplevel session : we first dynamically load the `graphics` module (`#load` directive) then dynamically compile and run (`#use` directive) a graphical Caml program (a color wheel).

4.2 A Chinese Sort

The screenshot of figure 7 illustrates builtin unicode capabilities of OCamIL together with direct calls to BCL static methods.

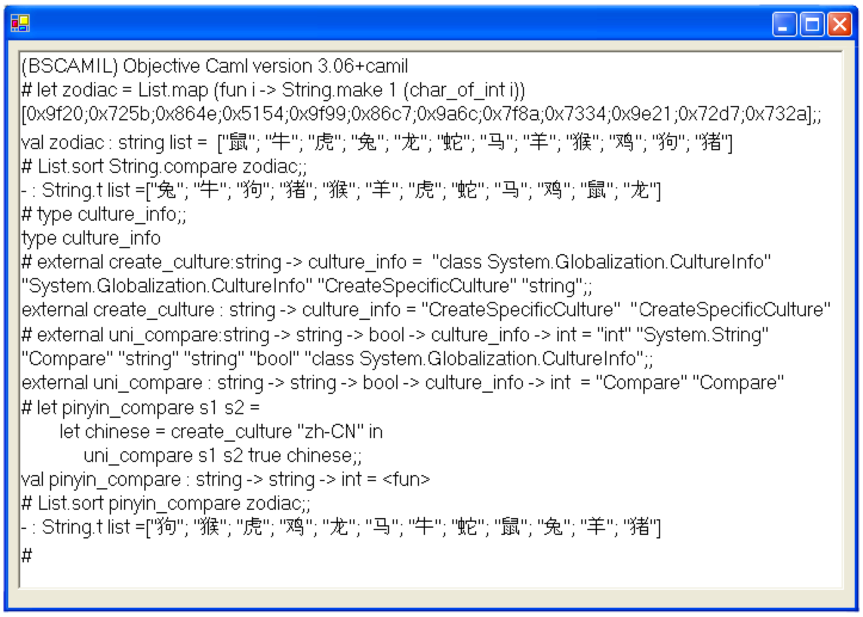

Fig. 7. Culture-specific ordering using external interface.

We define the list of characters representing the twelve chinese zodiac signs (rat, ox, tiger, rabbit, dragon, snake, horse, goat, monkey, rooster, dog and pig) by means of unicode codepoints. According to chinese pinyin[21] these are pronounced : *shu*, *niu*, *hu*, *tu*, *long*, *she*, *ma*, *yang*, *hou*, *ji*, *gou* and *zhu*.

Sorting the list using the standard comparison function `String.compare`, based on codepoints ordering, is quite meaningless, that is why we take advantage of methods provided by the .NET `System.Globalization` namespace in order to sort chinese characters according to pinyin.

[21] Pinyin is the official transliteration for Mandarin Chinese.

4.3 Benchmarks

The latest Windows Operating systems do not provide fully satisfying tools to measure execution times. We did not find any substitute for the Unix `time` command, which differenciates user and system times of a process. Using `time` under `cygwin`[22], a Unix layer for Windows only computes times for the main *thread*.

Morever JIT compilation introduces a difference between the first run of a program and the following runs. Therefore, we only measure the real-time of a .NET program execution.

Comparing OCamIL to Objective Caml bytecode compiler is informative. We also use two other compilers targeting .NET : F# which compiles the functional/imperative core of Objective Caml and SML.NET which compiles SML[3] core.

We test substantial programs such as `Boyer` (term computations, function calls), `KB` (a fully functional program using exceptions intensively to compute over terms) and `Nucleic` (floating-point calculations involving trees). The latter program is used in [4] to test a dozen of functional languages compilers.

The following benchmarks ran on a Windows XP Pentium IV 2,4GHz station. They are designed to run within a second under the native (`ocamlopt`) compiler.

	ocamlopt	ocamlc	OCamIL	F#	SML.NET
Boyer	0,42	1,92	31,9	28,0	24,7
KB	1,07	7,30	170	216	209
Nucleic	1,14	6,57	7,53	3,79	1,04

Fig. 8. Performance tests (real time in seconds).

Two trends appear :

– the three compilers that target .NET get poor results on fully functional programs (`KB` and `Boyer`).
– but results for monomorphic floating-points calculations are fairly similar.

F# compiles toward an extension of MSIL, called ILX [5] which introduces genericity. SML.NET, like MLj [6], analyses the whole program at link time and specializes polymorphic functions.

OCamIL retyping of the `Tlambda` intermediate language is not accurate enough, entailing costly data structure allocation (object arrays). Data access is then slower and needs a dynamic typechecking. To increase performance, OCamIL needs to retrieve more type information from the regular typing phase of Objective Caml. Compiling to a typed virtual machine raises new issues that were not relevant in dedicated functional virtual machines [7].

22 `www.cygwin.org`

4.4 The OCamIL Distribution

The first version of OCamIL is available [14] for Windows platform. We have ported the main part of Objective Caml standard library, as well as the Caml `graphics` library. Functional, imperative and object-oriented features[23] are implemented, as well as the module system (functors, modular compilation).

5 Related Works

Compiling program pieces written in different languages and targeting a single runtime is an old idea of the functional programming community. With a view towards interoperability *via* C, several compilers to C were designed at the beginning of the 90's (like [8], [9], [10]). Targeting a virtual machine runtime that manages memory and handles exceptions, such as the JVM and its associated runtime ensures a better level of safety during execution. This has greatly contributed to the success of the Java platform. MLj [6] and Bigloo [10] already compile statically or dynamically typed functional languages to Java bytecode.

The .NET platform enhancements are threefold : a specified type system, more accurate operators to manage stacks (for tail recursive calls and stack-allocated value types) and a more understandable JIT. In either case, it is rather difficult to compile some programming features that do not fit naturally in Java or C# object models, such as closures (ML, Scheme, Haskell), multiple inheritance (Eiffel, Objective Caml) and continuations (Scheme).

As described in the introduction, a number of compilers have been adapted to the .NET platform, but only a few have built a toplevel. For instance Bigloo uses its interpreter. As far as we know, only P# has followed the hard way to bootstrapping, by means of C# code generation.

The many interface definition languages (IDL) for CORBA[24] or for COM[25] have a similar goal. Some of them are designed for functional languages, such as HDIRECT[11] for Haskell[26] or OCAMLIDL[27] for Objective Caml.

6 Conclusion

Our experimental OCamIL compiler and toplevel allow the development of Objective Caml applications for the .NET platform, with the guarantee of compatibility with Objective Caml (including advanced programming features [12]) and managed MSIL code production. In particular, this allows us to embed our toplevel as a component inside a C# application.

Further work will enhance typing for generated MSIL code, introduce interfacing facilities with existing libraries and also take multi-threading management

[23] Check out OCAMIL webpage [14] for details.
[24] `www.omg.org`
[25] `www.microsoft.com/com`
[26] `www.haskell.org`
[27] `caml.inria.fr/camlidl`

into account.
Inferring more accurate type informations in our new intermediate language (`Tlambda`) will help to improve efficiency (less boxing/unboxing) and to explore values during debugging. We aim at communication between the Objective Caml and C# object models. To achieve it, we will propose an IDL corresponding to the intersection of both models. This way has been tested for Java and Objective Caml[28] but has encountered some difficulties because the two corresponding runtimes do not catch up very well (because of GC and threads). We plan to integrate the Objective Caml concurrency model inside .NET.

Acknowledgement

We would like to thank Clément Capel for his help to adapt Caml libraries.

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. Appel, A.: Runtime tags aren't necessary. Lisp and Symbolic Computation (1989)
3. Milner, R., Tofte, M., Harper, R.: The Definition of Standard ML. MIT Press, Cambridge, MA (1991)
4. Hartel, P.H., *et al*, M.F.: Benchmarking implementations of functional languages with "Pseudoknot", a float-intensive benchmark. Journal of Functional Programming **6** (1996) 621–655
5. Syme, D.: ILX: Extending the .NET common IL for functional language interoperability. Electronic Notes in Theoretical Computer Science **59** (2001)
6. Benton, N., Kennedy, A., Russel, G.: Compiling Standard ML to Java Bytecodes. In: Proceedings of the 3rd ACM SIGPLAN Conference on Functional Programming. (1998)
7. Leroy, X.: The effectiveness of type-based unboxing. In: Workshop on Types in Compilation. (1997)
8. Chailloux, E.: An Efficient Way of Compiling ML to C. In: Workshop on ML and its Applications, ACM SIGPLAN (1992)
9. Tarditi, D., Lee, P., Acharya, A.: No assembly required: Compiling standard ML to C. ACM Letters on Programming Languages and Systems **1** (1992) 161–177
10. Serrano, M., Weis, P.: Bigloo: a portable and optimizing compiler for strict functional languages. In: 2nd, Glasgow, Scotland (1995) 366–381
11. Finne, S., Leijen, D., Meijer, E., Jones, S.P.: H/direct: A binary foreign language interface for haskell. In: International Conference on Functional Programming. (1998)
12. Aponte, M.V., Chailloux, E., Cousineau, G., Manoury, P.: Advanced programming features in objective caml. In: 6th Brazilian Symposium on Programming Languages. (2002)

[28] `www.pps.jussieu.fr/~henry/ojacare`