Experience Report: Mirroring reFLect to OCaml

Michel Mauny
ENSTA
Michel.Mauny@ensta.fr

Nicolas Pouillard
INRIA Paris-Rocquencourt
Nicolas.Pouillard@inria.fr

Abstract

We report in this paper on our experience in the design and implementation of a tool for migrating reFLect code to OCaml. Migration proceeds by translating program parts to pure OCaml code, and binding the rest of the reFLect program as OCaml primitives. Linking both reFLect and OCaml runtime systems into a single executable allows running partially translated applications.

1. Introduction

Code migration from a programming language $\mathcal{S}$ to a language $\mathcal{T}$ is a difficult task. Ideally, it could consist in a translator encoding more or less directly the semantics of $\mathcal{S}$ programs into $\mathcal{T}$. If the resulting $\mathcal{T}$ code is meant to be readable, the encoding must be rather direct, which is possible when language $\mathcal{S}$ is close to a sub-language of $\mathcal{T}$.

For numerous reasons, designing a complete translator able to translate any $\mathcal{S}$ program into $\mathcal{T}$ is a task whith a rather high failure probability. Translating exhaustively all language features not only requires $\mathcal{S}$ to be completely defined, but may also involve important resources (manpower) for some exotic, rarely used features. Furthermore, $\mathcal{S}$ may provide a rather big primitive environment, whose re-implementation through $\mathcal{T}$'s foreign function interface (FFI) may require more resources than available.

Most systems that are actively used are also being modified and extended. This presents a problem for anyone who wants to move an application from one language to another because it requires that development be put on hold while the translation is completed. For large complex systems that could be a deal breaker as the translation is going to be time consuming.

Alternatively to such a complete translation project, one may provide a migration tool able to translate part of a complete $\mathcal{S}$ program P into $\mathcal{T}$, and to expose the rest of P as foreign primitives to the translated part. Linking together the $\mathcal{T}$ runtime system, the translated part of program P, the bindings of the un-translated part of P, and the $\mathcal{S}$ runtime system, one should be able to run the partially translated application. This is the path that we followed for designing a migration tool from the reFLect (Grundy et al. 2006) programming language to Objective Caml (Leroy et al. 2008).

We report in this paper our experience in the design and implementation of this migration tool. After the short description of the context of this project given in section 2, section 3 gives the architecture of the migration tool. The main characteristics of the translation process are described in section 4 and the generation of bindings is described in section 5. Section 6 discusses some of the aspects of the languages involved in this development that have been either useful or otherwise.

Submitted to ICFP 31 August – 2 September 2009, Edinburgh.

2. The project

The Forte (Seger et al. 2005) formal verification environment integrates model-checking tools and theorem proving in a general purpose lazy functional language named reFLect. The distinctive features of reFLect are the inclusion of binary decision diagrams (BDDs) as primitive objects generalizing boolean constants and operations, the availability of reflection operations used to construct and decompose expressions in the reFLect language itself, and an ML type discipline extended with a flexible overloading layer.

The goal of the project that we describe here is to provide a path for migrating reFLect code to OCaml. Fortunately, we did not need to write a parser and type-checker for reFLect. We have been able to reuse the front-end of a reFLect compiler built during another project. This compiler is named CaFL and has been designed and implemented by Virgile Prevosto and the authors, with the help of Damien Doligez.

The project aims at producing a migration tool that should be able to translate a reFLect file into a legal OCaml source file. We want to be able to run the whole program, even when the whole source has not yet been translated. This allows applications being migrated to reuse existing reFLect primitives, libraries and program parts, and thus adds flexibility to the migration process.

3. The big picture

As described above, this project needs the following components:

1. a translator from reFLect to OCaml, that encodes reFLect laziness into the (strict) OCaml language;
2. an interface generator that binds reFLect code to OCaml, making it available to OCaml as a set of primitives.

The flexibility of this tool comes from the possibility of progressively moving the frontier between binding and translation: we can translate part of a reFLect application, and bind the rest of its code to OCaml. Linking the code produced by the OCaml compiler to the reFLect runtime system enables running the application whose part has been translated.

Figure 1 sketches the code organization of the resulting, partly translated, applications. The translated part, in grey, is OCaml code that may call C primitives through OCaml's FFI. The un-translated part is made of reFLect code bound to OCaml as primitives, and using its own set of primitives written in C or C++, through reFLect's FFI .

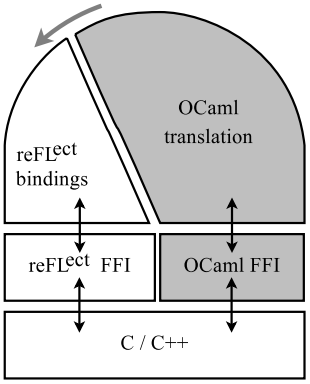

Figure 1. The code of a partially translated program. Double arrows denote direct interactions, and the grey arrow shows the migration progress.

When more reFLect gets translated to OCaml, less reFLect remains as primitive, augmenting the grey area on figure 1. A complete translation to OCaml of a reFLect application needs not only to translate all reFLect code, greying out all reFLect source code of the application, but also requires binding to OCaml *all* reFLect primitives written in C/C++ using OCaml's FFI, eliminating the usage of reFLect's FFI. The reFLect language has such a rich predefined environment—several thousands of primitives and library functions—that a complete translation to OCaml is not realistic.

4. Translation from reFLect to OCaml

The translation of reFLect programs to OCaml has to perform two main tasks: overloading resolution and encoding of laziness.

The CaFL front-end implements a complete reFLect parser and type-checker and produces desugared programs where overloading has been resolved. When an occurrence of an overloaded symbol can be statically resolved, it is immediately replaced by the corresponding instance of the symbol. When the static context does not provide enough information, the current definition receives an extra formal parameter that will eventually be instantiated at call sites with the actual value it denotes at that point. reFLect has named arguments and CaFL also erases names and reorders arguments. Finally, although CaFL processes reflection features, we chose to postpone their translation into OCaml to a later phase of the project.

The output of this parsing/typing front-end can be seen as an OCaml, implicitly lazy, source program. In the following, we write LOCaml for this "Lazy OCaml" language. The next step of the translation consists in making laziness explicit using essentially the following rules (Mauny 1991), which translate the core of LOCaml into OCaml:

$$\begin{array}{lcl} [\![e_1\ e_2]\!] & = & [\![e_1]\!](\texttt{lazy}\ [\![e_2]\!]) \\ [\![\lambda x.e]\!] & = & \lambda x.[\![e]\!] \\ [\![x]\!] & = & \texttt{Lazy.force}\ x \end{array}$$

This translation delays the computation of function arguments and forces the evaluation of the values resulting from looking up variables. These rules easily extend to a complete language, including data structures whose components evaluation is performed only when they are accessed.

The OCaml construct[1] (`lazy` e) has type (τ `Lazy.t`) when e has type τ, and (`Lazy.force` e) has type τ if e has type (τ `Lazy.t`). This translation scheme preserves typability: it changes legal, implicitly lazy, LOCaml source code to legal OCaml source code with explicit laziness.

Although it may look superfluous, the LOCaml intermediate step is important for the readability of the translation: the final OCaml translation of an original piece of reFLect is polluted by lots of `lazy` constructs and calls to `Lazy.force` and is itself very difficult to read. The LOCaml version can therefore be thought of as *the* translation, abstracting out the evaluation strategy. Getting closer to OCaml would consist in removing as much laziness as possible in translated programs, keeping only the laziness without which the program would crash or consume too much time or space. Adding—possibly manually—laziness annotations at appropriate places of a LOCaml source program might be a way of changing an implicitely lazy program into a mostly strict OCaml program, final result of a successful migration.

5. Binding reFLect to OCaml

As already mentioned, the amount of reFLect/C/C++ code to be rewritten to OCaml, as well as the amount of C/C++ code to bind directly to OCaml using OCaml's FFI was so big that one could not reasonably aim at a tool able to perform a full migration of reFLect applications. Binding reFLect to OCaml was therefore mandatory. We started with a basic low-level interface between reFLect and OCaml on top of which we designed a more sophisticated type-safe interface based on the structure of types of the reFLect values to be imported to OCaml.

5.1 At the low level

The OCaml and reFLect runtime systems are rather different from each other: the evaluation of reFLect uses graph reduction, and its memory manager uses reference counting techniques assisted by a mark and sweep collector for scavenging unreachable cycles and tracing memory blocks with saturated reference counts. OCaml, as a strict language, uses a more conventional execution model and performs memory recycling using tracing techniques.

Memory management Both reFLect and OCaml provide ways to inform their respective memory managers that pointers to local memory are used remotely, and both enable developers to register finalization procedures to be used when local memory has been remotely released. It has therefore been rather easy to establish and handle pointers to remote data from each of the memory spaces. We also knew that we could have to face problems for collecting free cyclic structures spanning over the two memory spaces (Plainfossé and Shapiro 1995), but we chose to ignore them at that point of the project, as long as they do not block the execution of translated code.

Evaluation The reFLect graph reduction and the encoding of laziness in OCaml each have their own invariants, and exporting to reFLect an OCaml suspension that involves reFLect graphs, handling themselves OCaml values, gave us quite a few headaches!

The translation of LOCaml to OCaml is such that `Lazy.force` produces an OCaml value (assuming no exception is raised) that is *not* a suspension. This can be checked by a mere inspection of the translation rules given at section 4. In other words, forcing *once* is sufficient. In reFLect, forcing the evaluation of a graph is supposed to produce a value in weak head normal form.

Extending those invariants to the reFLect/OCaml cooperation implies that:

- when a reFLect object (a graph) is imported to OCaml as a primitive, it must be represented as a suspension whose forcing should fire the evaluation of the graph on the reFLect side;
- when an OCaml suspension is exported to reFLect, it must be exposed as a graph to the reFLect graph reducer, and evaluating

[1] `lazy` is a keyword in OCaml.

that graph in reFLect should `Lazy.force` the OCaml suspension, if needed.

Importing and handling reFLect graphs Using reFLect dynamic features such as introspection of the runtime environment, we can list the reFLect primitives and (library or user-defined) functions that we may want to use from OCaml. Importing them effectively is then a matter of obtaining their value through the reFLect interpreter, and declaring them to OCaml as external data of a specific abstract type named `fl`.

Instead of using reFLect graphs from OCaml as untyped external data, we provide the abstract type `fl` with a type parameter. For instance, the type $((\alpha \to \alpha)\ \texttt{fl})$ is the (phantom) type of reFLect graphs that implement functions from α to α. Importing reFLect values to OCaml is performed during the initialization of the resulting program, immediately asserting the conformity of the expected OCaml type with the actual reFLect type computed by the reFLect interpreter.

5.2 A type-safe interface

With such statically typed data and a few primitive operations, it is possible to build a rather complete, type-safe, interface between reFLect and OCaml. For instance, if we know how to build application graphs from OCaml with a function of type $(\alpha \to \beta)\ \texttt{fl} \to \alpha\ \texttt{fl} \to \beta\ \texttt{fl}$, we are able to compute the application of an external function g_1 with OCaml type $(\tau_1 \to \tau_2)\ \texttt{fl}$ to an external argument $g_2 : \tau_1\ \texttt{fl}$.

A closer look at the type $(\alpha \to \beta)\ \texttt{fl} \to ((\alpha\ \texttt{fl}) \to (\beta\ \texttt{fl}))$ of this graph building function tells us that this function imports the (functionality denoted by the) arrow type constructor from reFLect to OCaml. This mechanism can be generalized to the whole reFLect type algebra: given primitives implementing the imported semantics of data constructors (*e.g.* how to represent in OCaml the reFLect list constructors), compute mechanically a function that changes a reFLect data of type $(\alpha\ \texttt{list})\ \texttt{fl}$ into an OCaml data of type $(\alpha\ \texttt{fl})\ \texttt{lazy\_list}$. This is a marshaling function automatically computed from the structure of types.

5.3 Code migration: translating more, binding less

We used this process to mechanically generate thousands of bindings. Clearly, such an automatic generation is both more efficient and less error-prone than manual binding of primitives. More importantly, it gives us the flexibility that we were looking for. Being able to bind arbitrary reFLect code to OCaml, and not only a fixed set of primitives, enables us to choose which part of the code is to be translated and which has to be bound as primitive operations.

6. Discussion

We list here some of the language features that have been important to the project.

6.1 Meta-programming features

Reflection/Introspection in the reFLect language The reFLect system provides ways to dynamically inspect its current global environment by giving access to the global symbol table: one can list all globally defined values and types, and for each of them, access some of their properties like their fixity (infix, postfix, *etc.*), their type, whether they are overloaded or not.... The generation of reFLect bindings uses these features to list the symbols to be bound to OCaml, and to compute at what type they will be usable by the OCaml part.

The interactive nature of reFLect and the availability of evaluation functions have also been of great help for the project. Interaction enables dynamic loading of code that performs the extraction of the current symbols to be bound into OCaml, and, while computing bindings, the type information inferred by the evaluation function is compared to the type expected on the OCaml side. This comparison is an assertion whose failure corresponds to an internal bug of our tool.

Syntax extensions using CamlP4 Another big win of meta-programming in this context is the heavy usage of compile time code transformations performed by sophisticated pre-processing techniques.

The OCaml system is shipped with a user-programmable preprocessor called CamlP4 whose aim is to extend the OCaml syntax with domain specific notations. Ranging from aesthetic shortcuts to new constructs involving complex code transformations or hiding unsafe function calls, the CamlP4 extensions that we used constitute a complete mechanism for the high-level specification of reFLect to OCaml bindings.

CamlP4 provides parsers and printers for the whole OCaml language that have been used when programming the translator from LOCaml to OCaml, which encodes laziness into OCaml.

6.2 The type algebra

The reFLect and OCaml type algebras have a common subset from which we established an automatic generation of marshaling functions, based on the type structure of the objects to be marshaled. The similarities between reFLect and OCaml types in this subset and the algebraic nature of the type structure enabled this automatic generation of "type-based cross-language marshaling" functions.

Still, reFLect has types or features that either have no correspondent in OCaml: overloading and named arguments of data constructors are eliminated by the CaFL front-end, and the types whose OCaml counterparts are either not obvious or simply not yet addressed are left as external (abstract) types.

6.3 Built-in OCaml laziness

The OCaml built-in constructs and primitives for encoding laziness are intensively used in this project.

In reFLect, top-level declarations may be polymorphic. Encoding their lazy nature with a mutable cell holding the computation of a polymorphic value would be forbidden by the ML type discipline. The OCaml (`lazy _`) construct is considered as *non-expansive*, just like a function abstraction, and its type can safely be generalized.

The translation scheme that we use (see section 4) generates lots of suspensions and calls to `Lazy.force`. Creating a suspension holding an expression is operationally useless when the expression is already a value (for instance a constant or an abstraction). For efficiency reasons, such suspensions should be avoided, but naively omitting the `lazy` construct is not possible because it would break the typability of the whole program.

The OCaml runtime system represents suspensions as blocks holding a special tag `lazy_tag`. When a suspension is forced into a value, its original block gets updated with the value and becomes an indirection block by changing its tag into `forward_tag`. Since indirection blocks are immutable and admit only one operation (`Lazy.force`), they may safely be eliminated by the garbage-collector, as long as sharing is preserved and `Lazy.force` is aware of all possible representation of evaluated lazy values (Doligez 2008). It is therefore safe to compile $(\texttt{lazy}\ e)$ as e when it is safe to evaluate e at this point of the program. Currently, when e is a constant, an identifier or an abstraction, then

- if its type cannot be `float`[2] or $(\tau\ \texttt{Lazy.t})$ for some τ, $(\texttt{lazy}\ e)$ gets compiled as e;

[2] For optimizing data structures holding floating point numbers, the OCaml compiler assumes that only expressions of type `float` can have a `float` representation.

- otherwise, we generate code for computing the value of e and put it in an indirection box.

Making laziness explicit also produces lots of calls to `Lazy.force` that first test the tag of their argument, and according to the tag either force the suspension or perform a projection. We modified the OCaml compiler in order to inline the tag tests, and project the value whenever possible. This way, a function call is performed only when there is some real work to be done.

We introduced these optimizations in the OCaml 3.11 compiler. This, together with the special typing of suspensions, simplified a lot the task of encoding a lazy ML in OCaml.

6.4 Compilers don't like generated programs

Program generation is one of those subjects about which compiler writers and intensive users may quarrel. Should a compiler do its best to "swallow" any program, or should program generators always remain under the compilers' limits? The first option is a nightmare for compiler writers: they prefer to concentrate their efforts on good code generation rather than be able to compile programs that no human being could have produced "by hand". The second option looks reasonable, but program generators often have to guess where the limits are and in what way they depend on the current architecture.

Of course, we have been bitten by such limits of the OCaml environment. The OCaml parser provided by CamlP4 processes source files in one shot, and big files hit the limits enforced by the operating system. In order to perform global optimizations, the amount of stack space used by one of the native code OCaml compilers depend on the size of their input file.

The solution is clearly to design the translator in such a way that it produces outputs of reasonable size, and to warn its users that they should translate and bind reFLect code in a modular way, whenever possible.

6.5 OCaml and reFLect runtime cooperation

Establishing a tight cooperation between the OCaml and reFLect runtime systems has been a crucial step in our project. However, the legend says that memory management is tricky and error-prone, and that bugs are difficult to track and identify. Interfacing the reFLect and OCaml memory managers did not contradict this belief!

OCaml and reFLect provide similar ways of informing their respective garbage collectors (GCs) that memory blocks pointed to by local C/C++ variables should not be recycled. This management of roots is sufficient to be safe for tracing techniques, where unreachability is the only criterion used for releasing memory: at the C level, once global and local pointers to memory blocks are declared as roots, one may safely concentrate on the actual computations to be programmed.

On the other hand, the correctness of reference counters with respect to effective reachability is a permanent issue: counters have to be adjusted during the computations, and forgetting to increment one of them quickly leads to a disaster.

Even with their well-known weaknesses, reference counting techniques remain intellectually pleasant and seem easy to work with. On the other hand, generational copying collectors are known to be efficient, and well adapted to functional languages and to current hardware. Still, programmers may worry about handling pointers to data that are moved by the GC. Our experience confirms that in fact, reference counting needs more management than tracing techniques, which impose only the requirement to inform the GC where the memory roots are, and to make sure that only indirect pointers to memory are used when a copying GC may occur.

7. Conclusion

We reported in this paper on the design and implementation of a migration tool from reFLect to OCaml. Even though both languages share major features such as higher-order functions and a static ML-like typing discipline, migrating *real* code from reFLect to OCaml is not that easy. Partial translation and runtime cooperation may represent a reasonable migration path. Obviously, a deep knowledge of both languages as well as of their implementations is necessary for the success of such a project. Furthermore, language features such as meta-programming tools, the ML type discipline and the algebraic nature of its type algebra, built-in alternative evaluation strategies, and foreign function interfaces have been, among others, invaluable tools.

References

Damien Doligez. Lazy evaluation. in (Leroy et al. 2008), 2008.

Jim Grundy, Tom Melham, and John O'leary. A reflective functional language for hardware design and theorem proving. *J. Funct. Program.*, 16(2):157–196, 2006. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796805005757.

Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon). *The Objective Caml system, release 3.11*, 2008. URL `http://caml.inria.fr/distrib/ocaml-3.11/ocaml-3.11-reman.pdf`.

Michel Mauny. Integrating lazy evaluation in strict ML. Technical Report 137, INRIA, 1991. URL `http://www.mauny.net/data/papers/mauny-1992a.pdf`.

David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In Henry G. Baker, editor, *Proc. Int. Workshop on Memory Management (IWMM)*, volume 986 of *Lectore Notes in Computer Science*, pages 211–249, Kinross, Scotland (UK), September 1995. Springer Verlag.

C.-J.H. Seger, R.B. Jones, J.W. O'Leary, T. Melham, M.D. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(9):1381–1405, September 2005. ISSN 0278-0070. doi: 10.1109/TCAD.2005.850814.