22
Developing applications with Objective Caml

Having reached this point, the reader should no longer doubt the richness of Objective Caml. This language rests on a functional and imperative core, and it integrates the two major application organization models: modules and objects. While presented as libraries, threads are an attractive part of the language. The system primitives, portable for the most part, complete the language with all the possibilities offered by distributed programming. These different programming paradigms are shaped within the general framework of static typing with inference. For all that, these elements do not, in themselves, settle the question of Objective Caml's relevance for developing applications, or more prosaically, "is it a good language?"

None of the following classic arguments can be used in its favor:

- (marketing development) "it's a good language because clients buy it";
- (historical development) "it's a good language because thousands of lines of code have already been written in it";
- (systems development) "it's a good language because the Unix or Windows systems are written in it";
- (beacon application development) "it's a good language because such-and-such application is written in it";
- (standardization development) "it's a good language because it has an ISO specification".

We'll review one last time the various features of the language, but this time from the angle of its relevance for answering a development team's needs. The criteria selected to make up the elements of our evaluation take into account the intrinsic qualities of the language, its development environment, the contributions of its community and the significant applications which have been achieved. Finally we'll compare Objective Caml with several similar functional languages as well as the object-oriented language Java in order to underscore the main differences.

Elements of the evaluation

The Objective Caml distribution supplies two online compilers, one generating bytecodes and the other producing instructions for the most modern processors. The toplevel uses the bytecode compiler. Beyond that, the distribution offers numerous libraries in the form of modules and some tools for calculating dependencies between modules, for profiling, and for debugging. Finally, thanks to its interface with the C language, it is possible to link Objective Caml programs to C programs. Languages which similarly offer an interface with C, such as the JNI (Java Native Interface) of Java, become accessible in this way.

The reference manual gives the language syntax, and describes the development tools and the library signatures. This set (language, tools, libraries, documentation) makes up a development environment.

Language

Specification and implementation

There are two ways to approach a new language. A first way is to read the language specification to have a global vision. A second is to plunge into the language's user manual, following the concepts illustrated by the examples. Objective Caml has neither one, which makes a self-taught approach to it relatively difficult. The absence of a formal specification (such as SML has) or a descriptive one (such as ADA's) is a handicap for understanding how a program works. Another consequence of the lack of a specification is the impossibility of getting a language standard issued by ISO, ANSI, or IEEE. This strongly limits the construction of new implementations tailored for other environments. Fortunately INRIA's implementation is of high quality and best of all, the sources of the distribution can be downloaded.

Syntax

The particulars of syntax are a non-negligible difficulty in approaching Objective Caml. They can be explained by the functional origin of the language, but also by historical, that is to say anecdotal, factors.

The syntax of application of a function to its arguments is defined by simple juxtaposition, as in `f 1 2`. The lack of parentheses bothers the neophyte as much as the C programmer or the confirmed Lisp programmer. Nevertheless, this difficulty only arises when reading the code of a programmer who's stingy with parentheses. Nothing stops the neophyte Objective Caml programmer from using more explicit parenthesization and writing `(f 1 2)`.

Beyond the functional core, Objective Caml adopts a syntax sometimes at odds with customary usage: access to array elements uses the notation `t.(i)` and not the usual brackets; method invocation is noted by a pound (# character) and not a dot, etc. These idiosyncrasies don't make it easier to get a grip on Objective Caml.

Finally, the syntax of Objective Caml and its ancestor Caml has undergone numerous modifications since their first implementation. Which hasn't pleaded in favor of the enduring nature of the developed applications.

To end on a more positive note, the pattern-matching syntax, inherited from the ML family, which Objective Caml incorporates, structures function definitions by case pleasingly and with simplicity.

Static typing

The fundamental character of the Objective Caml language resides in the language's static typing of expressions and declarations. This guarantees that no type error surfaces during program execution. Static type inference was conceived for the functional languages of the ML family, and Objective Caml has been able to maintain the greater part of this type discipline for the imperative and object-oriented extensions. However, in the object-oriented case, the programmer must sometimes give type inference a hand through explicit type constraints. Still, Objective Caml preserves static typing of expressions, and definitions, which provides an unsurpassed measure of execution safety: an Objective Caml program, will not return the "method not found" exception, which is not the case for dynamically typed object-oriented languages.

Objective Caml's parametric polymorphism of types allows the implementation of general algorithms. It is channeled into the object-oriented layer where parametric classes produce generic code, and not an expansion of code as generated by the templates of other languages. In itself, parametric polymorphism is an important component of code reusability.

The object-oriented extension adds a notion of inclusion polymorphism which is obtained by specifying subtyping relationships between objects. It reconciles code reusability, which constitutes the strength of the inheritance relationship between classes, with the security of static typing.

Libraries and tools

The libraries supplied with the distribution cover great needs. The programmer finds there, as a standard, the implementation of the most usual data structures with their basic functions. For example: stacks, queues, hash tables, AVL trees. More advanced tools can be found there as well, such as the treatment of data streams. These libraries are enriched in the course of successive language versions.

The `Unix` library allows lower-level programming as much for I/O as for process management. It is not identical for all platforms, which limits its use to some extent.

The exact arithmetic library and the regular expression library facilitate the development of specific applications.

Unfortunately the portable graphics library offers little functionality to support the construction of graphical user interfaces.

C libraries can easily be interfaced with the Objective Caml language. Here, the free availability of well-structured and duly commented sources definitely unlocks the potential for contact with the outside world and the various and sundry libraries to be found there.

Among the supplied tools, those for lexical and syntactic analysis, indispensable whenever dealing with complex textual data, are especially noteworthy. Based on the classic `lex` and `yacc`, they integrate perfectly with sum types and the functionality of Objective Caml, thus making them simpler to use than their predecessors.

Finally, no matter what soundness its features may bring, use of a language "in actual practice" never avoids the debugging phase of a program. The Objective Caml 2.04 distribution does not supply an IDE. Certainly, using the toplevel allows one to proceed rapidly to compilation and unit tests of functions (which is an undeniable advantage), but this usage will vary by platform and by programmer: cut-and-paste under X-Windows, calling the shell under `emacs`, requesting evaluation of a buffer under Windows. The next version (see appendix B) provides for the first time an environment for Unix containing a browser for interfaces and modules and a structured editor linked to the toplevel. Finally the distribution's debugger remains hard to use (in particular, because of the functional aspect and the language's parametric polymorphism) and limited to the Unix system.

Documentation

The documentation of the distribution consists of the reference manual in printable (PostScript) and online (HTML) format. This manual is not in any case an introduction to the language. On the contrary it is indispensable for finding out what's in the libraries or what parameters the commands take. Some pedagogical material can be found on INRIA's Caml page, but it is mostly regarding the Caml-Light language. Thus the language is missing a complete tutorial manual, we hope that this book fills this gap.

Other development tools

We have restricted ourselves up to now to the Objective Caml distribution. Nevertheless the community of developers using this language is active, as demonstrated by the number of messages on the `caml-list@inria.fr` mailing list. Numerous tools, libraries, and extensions are used to facilitate development. In the following we detail the use of tools for editing, syntax extension, interfacing with other languages and parallel programming. We mention as well the numerous graphical interface libraries. Most of these contributions can be found on the "Caml Hump" site:

Link: http://caml.inria.fr/hump.html

Editing tools

There are several modes recognizing Objective Caml syntax for the `emacs` editor. These modes are used to automatically indent text in the course of entering it, making it more readable. This is an alternative to the interaction window under Windows. Since `emacs` runs under Windows, the Objective Caml toplevel can be launched within one of its windows.

Syntax extension

The lexical and syntactic analysis tools provided by the distribution are already quite complete, but they don't support extending the syntax of the language itself. The `camlp4` tool (see the link on page 313) is used in place and instead of Objective Caml's syntactic analyzer. The latter's compilers have only to proceed to typing and code generation. This tool allows the user to extend the syntax of the language, or to change to the original syntax. Moreover it offers pretty-printing facilities for the generated syntax trees. In this way it becomes easy to write a new toplevel for any Objective Caml syntax extension, or even another language implemented in Objective Caml.

Interoperability with other languages

Chapter 12 detailed how to interface the Objective Caml language with C. A multi-language application takes advantage of the features of each one, all while making different codes sharing a single memory space live in harmony. Nevertheless encapsulating C functions to make them callable from Objective Caml requires some tedious work. To simplify it, the `camlIDL` tool (see the link on page 350) supplies an interface generator and tools for importing COM (Windows) components into Objective Caml. The interfaces are generated from an `IDL` interface description file.

Graphical interfaces

The `Graphics` library allows the development of drawings and simple interactions, but it can't be considered a graphical interface worthy of the name. Chapter 13 has shown how this library could be extended to construct graphical components responding to some interactions. Using `Graphics` as a base allows us to preserve the portability of the interface between different platforms (X-Windows, Windows, MacOS), but limits its use to the low level of events and graphics contexts.

Several projects attempt to fill this gap, unfortunately none succeeds in being complete, portable, documented, and simple to use. Here is a list (extracted from the "Caml Hump") of the main projects:

- OlibRt: under this sweet name, it is a veritable toolbox, constructed under X-Windows, but not documented. Its distribution contains complete examples and in particular numerous games.
  Link: http://cristal.inria.fr/~ddr/

- camlTk is a complete and well documented interface to the Tk toolkit. Its weak point is its dependency on particular versions of Tcl/Tk, which makes it difficult to install. It was used to build the web browser `mmm` [Rou96] written in Objective Caml.
  Link: http://caml.inria.fr/˜rouaix/camltk-readme.html
- The `Xlib` library has been rewritten in Objective Caml. `Efuns`, a mini-clone of `emacs`, was developed using it. `Xlib` is not really a toolbox, and is not portable to graphical systems other than X-Windows.
- mlGtk is an interface built on Gtk. It is in development and has no documentation. Its interest lies in being portable under Unix and Windows (because Gtk is) in a simpler fashion than Tk. Besides it uses Objective Caml's object-oriented layer—which doesn't happen without sometimes posing some problems.
- LabTk is an interface to Tcl/Tk, for Unix, using extensions to Objective Caml which will be integrated into the next version (see appendix B). It includes its own Tcl/Tk distribution which installs easily.

Despite the efforts of the community, there is a real lack of tools for constructing portable interfaces. It may be hoped that LabTk becomes portable to different systems.

Parallel programming and distribution

Threads and sockets already offer basic mechanisms for concurrent and distributed programming. Interfacing with the C language allows the use of classic parallel programming libraries. The only thing missing is an interface with CORBA for invoking methods of remote objects. On the other hand, there are numerous libraries and language extensions which use different models of parallelism.

Libraries

The two main parallel programming libraries, `MPI` (Message Passing Interface) and `PVM` (Parallel Virtual Machine), are interfaced with Objective Caml. Documentation, links, and sources for these libraries can be found on the site

Link: http://www.netlib.org

The "Caml Hump" contains the various HTTP addresses from which the versions interfaced with Objective Caml can be downloaded.

Extensions

Numerous parallel extensions of Caml-Light or Objective Caml have been developed:

- Caml-Flight ([FC95]) is a SPMD (Simple Program Multiple Data) extension of the Caml-Light language. A program executes a copy of itself on a fixed number of processes. Communications are explicit, there is only one communication operation `get` which can only be executed from within the synchronization operation `sync`.

Link: http://www.univ-orleans.fr/SCIENCES/LIFO/Members/ghains/caml-flight.html

- BSML [BLH00] is an extension by BSP operations. The language preserves compositionality and allows precise predictions of performance if the number of processors is fixed.
  Link: http://www.univ-orleans.fr/SCIENCES/LIFO/Members/loulergu/bsml.html
- OCAMLP3 [DDLP98] is a parallel programming environment based on the skeleton model of the P3L language. The various predefined skeletons can overlap. Programs may be tested either in sequential mode or parallel mode, thus supporting reuse of Objective Caml's own tools.
  Link: http://www.di.unipi.it/~susanna/projects.html
- JoCAML [CL99] is based on the join-calculus model which supports high-level operations for concurrency, communication, and synchronization in the presence of distributed objects and mobile code, all while preserving automatic memory management.
  Link: http://pauillac.inria.fr/jocaml/
- Lucid Synchrone ([CP95]) is a language dedicated to the implemenation of reactive systems. It combines the functionality of Objective Caml and the features of *data-flow synchronous* languages.
  Link: http://www-spi.lip6.fr/~pouzet/lucid-synchrone/

Applications developed in Objective Caml

A certain number of applications have been developed in Objective Caml. We will only speak of "public" applications, that is, those which anyone can use either freely or by buying them.

Like other functional languages, Objective Caml is a good compiler implementation language. The bootstrap[1] of the `ocaml` compiler is a convincing example. As well, numerous language extensions have been contributed, as seen previously for parallel programming, but also for typing such as O'Labl (part of which is in the process of being integrated into Objective Caml, see appendix B) or for physical units. Links to these applications can be found on the "Caml Hump".

Objective Caml's second specialty concerns proof assistants. The major development in this area is the program Coq which accompanies the evolution of Caml almost since its origin. Historically, ML was conceived as the command language of the LCF (Logic for Computable Functions) system, before becoming independent of this application. It is thus natural to find it as the implementation language of an important theorem-proving program.

1. Bootstrapping is the compilation of a compiler by the compiler itself. Arrival at a fixed point, that is to say the compiler and the generated executable are identical, is a good test of compiler correctness.

A third application domain concerns parallelism (see page 684) and communication of which a good example is the `Ensemble` system.

Link: http://www.cs.cornell.edu/Info/Projects/Ensemble/

A list, not exhaustive, of significant applications developed in Objective Caml is maintained on Inria's Caml site:

Link: http://caml.inria.fr/users_programs-eng.html

Let us mention in particular **hevea** which is a LaTeX to HTML translator which we have used to create the HTML version of this book found on the accompanying CD-ROM.

Link: http://pauillac.inria.fr/~maranget/hevea/

While of importance, the applications we've just mentioned don't represent what, at the beginning of this chapter, we christened a "beacon application". Moreover, they don't explore a new specialized domain demonstrating the relevance of using Objective Caml. It is not clear that this example can be issued from academia. It is more likely that it will come from industry, whether in conjunction with language standardization (and so its formal specification), or for the needs of applications having to integrate different programming and program organization styles.

Similar functional languages

There are several languages similar to Objective Caml, whether through the functional aspect, or through typing. Objective Caml is descended from the ML family, and thus it has cousins of which the closest are across the Atlantic and across the channel in the lineage of SML (Standard ML). The Lisp family, and in particular the Scheme language, differs from ML mainly by its dynamic typing. Two lazy languages, Miranda and Haskell, take up or extend ML's typing in the framework of delayed evaluation. Two functional languages, Erlang and SCOL, developed by the Ericsson and Cryo-Networks corporations respectively, are directed towards communication.

ML family

The ML family comprises two main branches: Caml (Categorical Abstract Machine Language) and its derivatives Caml-Light and Objective Caml, SML (Standard ML) and its descendants SML/NJ and mosml. Caml, the ancestor, was developed between 1986 and 1990 by INRIA's FORMEL project in collaboration with University Paris 7 and the École Normale Supérieure. Its implementation was based on Le_Lisp's runtime. It integrated within the language the definition of grammars and pretty-printers, which allowed communication of values between the language described and Caml. Its type system was more restrictive for mutable values, insofar as it did not allow such values to be polymorphic. Its first descendant, Caml-Light, no longer used the CAM machine, but instead used Zinc for its implementation. The name was nevertheless retained to show its ancestry. It contributed a more lightweight implementation while optimizing the allocation of closures and using an optimizing GC as a precursor to the actual GC. This streamlining allowed it to be used on the PC's of the time. The various Caml-Light versions evolved towards actual typing of imperative features and were enriched with numerous libraries. The following offshoot, Caml Special Light or CSL, introduced parameterized modules and a native-code compiler. Finally the baby is actually Objective Caml which mainly adds the object-oriented extension to CSL. Since there has never been a complete specification of the Caml languages, these various changes have been able to take place in complete freedom.

The SML approach has been the opposite. The formal specification [MTH97] was given before the first implementation. It is difficult to read, and a second book gives a commentary ([MT91]) on it. This method, specification then implementation, has allowed the development of several implementations, of which the best-known is SML/NJ (Standard ML of New Jersey) from Lucent (ex-AT&T). Since its origin, SML has integrated parameterized modules. Its initial type system was different from that of Caml for imperative features, introducing a level of weakness for type variables. The differences between the two languages are detailed in [CKL96]. These differences are being effaced with time. The two families have the same type system for the functional and imperative core, Objective Caml now has parameterized modules. SML has also undergone changes, bringing it closer to Objective Caml, such as for record types. If the two languages don't merge, this mainly derives from their separate development. It is to be noted that there is a commercial development environment for SML, MLWorks, from Harlequin:

Link: http://www.harlequin.com/products/

An SML implementation, `mosml`, based on Caml-Light's runtime, has also been implemented.

Scheme

The Scheme language (1975) is a dialect of the Lisp language (1960). It has been standardized (IEEE Std 1178-1990). It is a functional language with strict evaluation, equipped with imperative features, dynamically typed. Its syntax is regular and particular about the use of parentheses. The principal data structure is the dotted pair (equivalent to an ML pair) with which possibly heterogeneous lists are constructed. The main loop of a Scheme toplevel is written `(print (eval (read)))`. The `read` function reads standard input and constructs a Scheme expression. The `eval` function evaluates the constructed expression and the `print` function prints the result. Scheme has a very useful macro-expansion system which, in association with the `eval` function, permits the easy construction of language extensions. It not only supports interrupting computation (exceptions) but also resuming computation thanks to continuations. A continuation corresponds to a point of computation. The special form `call_cc` launches a computation with the possibility of resuming this one at the level of the current continuation, that is to say of returning to this computation. There are many Scheme implementations. It is even used as a macro language for the GIMP image manipulation software. Scheme is an excellent experimental laboratory for the implementation of new sequential or parallel programming concepts (thanks to continuations).

Languages with delayed evaluation

In contrast with ML or Lisp, languages with delayed evaluation do not compute the parameters of function calls when they are passed, but when evaluation of the body of the function requires it. There is a “lazy” version of ML called Lazy ML but the main representatives of this language family are Miranda and Haskell.

Miranda

Miranda([Tur85]) is a pure functional language. That is to say, without side effects. A Miranda program is a sequence of equations defining functions and data structures.

Link:
http://www.engin.umd.umich.edu/CIS/course.des/cis400/miranda/miranda.html

For example the fib function is defined in this way:

```
 fib a = 1, a=0
       = 1, a=1
       = fib(a-1) + fib(a-2), a>1
```

Equations are chosen either through guards (conditional expressions) as above, or by pattern-matching as in the example below:

```
fib 0 = 1
fib 1 = 1
fib a = fib(a-1)+ fib(a-2)
```

These two methods can be mixed.

Functions are higher order and can be partially evaluated. Evaluation is lazy, no subexpression is computed until the moment when its value becomes necessary. Thus, Miranda lists are naturally streams.

Miranda has a concise syntax for infinite structures (lists, sets): `[1..]` represents the list of all the natural numbers. The list of values of the Fibonacci function is written briefly: `fibs = [a | (a,b) <- (1,1),(b,a+b)..]`. Since values are only computed when used, the declaration of `fibs` costs nothing.

Miranda is strongly typed, using a Hindley-Milner type system. Its type discipline is essentially the same as ML’s. It accepts the definition of data by the user.

Miranda is the archetype of pure lazy functional languages.

Haskell

The main Haskell language website contains reports of the definition of the language and its libraries, as well as its main implementations.

Link: http://www.haskell.org

Several books are dedicated to functional programming in Haskell, one of the most recent is [Tho99].

This is a language which incorporates almost all of the new concepts of functional languages. It is pure (without side effects), lazy (not strict), equipped with an *ad hoc* polymorphism (for overloading) as well as parametric polymorphism *à la* ML.

Ad hoc **polymorphism** This system is different from the polymorphism seen up to now. In ML a polymorphic function disregards its polymorphic arguments. The treatment is identical for all types. In Haskell it is the opposite. A polymorphic function may have a different behavior according to the type of its polymorphic arguments. This allows function overloading.

The basic idea is to define type classes which group together sets of overloaded functions. A class declaration defines a new class and the operations which it permits. A (class) instance declaration indicates that a certain type is an instance of some class. It includes the definition of the overloaded operations of this class for this type.

For example the `Num` class has the following declaration:

```
class Num a where
  (+)    :: a -> a -> a
  negate :: a -> a
```

Now an instance `Int` of the class `Num` can be declared in this way:

```
instance Num Int where
  x + y    =  addInt x y
  negate x = negateInt x
```

And the instance `Float`:

```
instance Num Float where
  x + y    =  addFloat x y
  negate x = negateFloat x
```

The application of `negate Num` will have a different behavior if the argument is an instance of `Int` or `Float`.

The other advantage of classes derives from inheritance between classes. The descendant class recovers the functions declared by its ancestor. Its instances can modify their behavior.

Other characteristics The other characteristics of the Haskell language are mainly the following:

- a purely functional I/O system using *monads*;
- arrays are built lazily;
- views permit different representations of a single data type.

In fact it contains just about all the high-strung features born of research in the functional language domain. This is its advantage and its disadvantage.

Communication languages

ERLANG

ERLANG is a dynamically typed functional language for concurrent programming. It was developed by the Ericsson corporation in the context of telecommunications applications. It is now open source. The main site for accessing the language is the following:

Link: http://www.erlang.org

It was conceived so that the creation of processes and their communication might be easy. Communications take place by message passing and they can be submitted with delays. It is easy to define protocols via ports. Each process possesses its own definition dictionary. Error management uses an exception mechanism and signals can propagate among processes. Numerous telephony applications have been developed in Erlang, yielding non-negligible savings of development time.

SCOL

The SCOL language is a communication language for constructing 3D worlds. It was developed by the Cryo Networks corporation:

Link: http://www.cryo-networks.com

Its core is close to that of Caml: it is functional, statically typed, parametrically polymorphic with type inference. It is "multimedia" thanks to its API's for sound, 2D, and 3D. The 3D engine is very efficient. SCOL's originality comes from communication between virtual machines by means of channels. A channel is an (environment, network link) pair. The link is a (TCP or UDP) socket.

SCOL's originality lies in having resolved simply the problem of securing downloaded code: only the text of programs circulates on the network. The receiving machine types the passed program, then executes it, guaranteeing that the code produced does indeed come from an official compiler. To implement such a solution, without sacrificing speed of transmission and reception, the choice of a statically typed functional language was imposed by the conciseness of source code which it supports.

Object-oriented languages: comparison with Java

Although Objective Caml sprang from the functional world, it is necessary to compare its object-oriented extension to an important representative of the object-oriented languages. We pick the Java language which, while similar from the point of view of its implementation, differs strongly in its object model and its type system.

The Java language is an object-oriented language developed by the SUN corporation. The main site for access to the language is the following:

Link: http://java.sun.com

Main characteristics

The Java language is a language with classes. Inheritance is simple and allows redefinition or overloading of inherited methods. Typing is static. An inherited class is in a subtyping relationship with its ancestor class.

Java does not have parameterized classes. One gets two types of polymorphism: *ad hoc* by overloading, and of inclusion by redefinition.

It is multi-threading and supports the development of distributed application whether using sockets or by invoking methods of remote objects (Remote Method Invocation).

The principles of its implementation are close to those of Objective Caml. A Java program is compiled to a virtual machine (JVM). The loading of code is dynamic. The code produced is independent of machine architectures, being interpreted by a virtual machine. The basic datatypes are specified in such a way as to guarantee the same representation on all architectures. The runtime is equipped with a GC.

Java has important class libraries (around 600 with the JDK, which are supplemented by many independent developments). The main libraries concern graphical interfaces and I/O operations integrating communication between machines.

Differences with Objective Caml

The main differences between Java and Objective Caml come from their type system, from redefinition and from overloading of methods. Redefinition of an inherited method must use parameters of exactly the same type. Method overloading supports switching the method to use according to the types of the method call parameters. In the following example class B inherits from class A. Class B redefines the first version of the to_string method, but overloads the second version. Moreover the eq method is overloaded since the type of the parameter (here B) is not equal to the type of the parameter of the inherited method (here A). In the end class B has two eq methods and two to_string methods.

```
class A {
  boolean eq (A o) {  return true;}
  String to_string (int n ) { }
}

class B extends A {
  boolean eq (B o) {  return true;}
  String to_string (int n ) { }
  String to_string (float x, float y)
}
```

Although binding is late, overload resolution, that is determination of the type of the method to use, is carried out on compilation.

The second important difference derives from the possibility of casting the type of an object, as would be done in C. In the following example, two objects a and b are defined, of class A and B respectively. Then three variables c, d and e are declared while imposing a type constraint on the affected values.

```
{
  A a = new A ();
  B b = new B ();
  A c = (A) b;
  B d = (B) c;
  B e = (B) a;
}
```

Since the type of b is a subtype of the type of a, the cast from b to c is accepted. In this case the type constraint may be omitted. On the other hand the two following constraints require a dynamic type test to be carried out to guarantee that the values in c and in a do in fact correspond to objects of class B. In this program this is true for c, but false for a. So this last case raises an exception. While this is useful, in particular for graphical interfaces, these type constraints can lead to exceptions being raised during exception due to erroneous use of types. In this Java is a language typed partly statically and partly dynamically. Moreover the absence of parameterized classes quite often obliges one to use this feature to write generic classes.

Future of Objective Caml development

It is difficult for a new language to exist if it is not accompanied by the important development of an application (like Unix for C) or considerable commercial and industrial support (like SUN for JAVA). The intrinsic qualities of the language are rarely enough. Objective Caml has numerous qualities and some defects which we have described in the course of this chapter. For its part, Objective Caml is sustained by INRIA where it was conceived and implemented in the bosom of the CRISTAL project. Born of academic research, Objective Caml is used there as an experimental laboratory for testing new programming paradigms, and an implementation language. It is widely taught in various university programs and preparatory classes. Several thousand students each year learn the concepts of the language and practice it. In this way the Objective Caml language has an important place in the academic world. The teaching of computer science, in France, but also in the United States, creates numerous programmers in this language on a practical as well as a theoretical level.

On the other hand, in industry the movement is less dynamic. To our knowledge, there is not a single commercial application, developed in Objective Caml, sold to the general public and advertising its use of Objective Caml. The only example coming close is that of the SCOL language from Cryo-Networks. There is however a slight agitation in this direction. The first appeals for funding for Objective Caml application startups are appearing. Without hoping for a rapid snowball effect, it is significant that a demand exists for this type of language. And without hoping for a very short-term return on investment either, it is important to take notice of it.

It is now for the language and its development environment to show their relevance. To accompany this phenomenon, it is no doubt necessary to provide certain guarantees as to the evolution of the language. In this capacity, Objective Caml is only just now emerging and must make the choice to venture further out of academia. But this "entry into the world" will only take place if certain rules are followed:

- guaranteeing the survival of developments by assuring upward compatibility in future versions of the language (the difficulty being stability of new elements (objects, etc.));
- specifying the language in conjunction with real developers with a view to future standardization (which will permit the development of several implementations to guarantee the existence of several solutions);
- conceiving a development environment containing a portable graphical interface, a CORBA bus, database interfaces, and especially a more congenial debugging environment.

Some of the points brought up, in particular standardization, can remain within the jurisdiction of academia. Others are only of advantage to industry. Thus everthing will depend on their degree of cooperation. There is a precedent demonstrating that a language can be "free" and still be commercially maintained, as this was the case for the `gnat` compiler of the ADA language and the ACT corporation.

Link: http://www.act-europe.fr