An optimizing ML to C compiler *

Régis Cridlig

Ecole Normale Supérieure †
& INRIA-Rocquencourt

Abstract

Since the C language is a machine independent low-level language, it is well-suited as a portable target language for the implementation of higher-order programming languages. This paper presents an efficient C code generator for Caml-Light, a variant of CAML designed at INRIA. Fundamentally, the compilation technique consists of translating ML code via an intermediate language named Sqil and the runtime system relies on a new conservative garbage collector. This scheme produces at the same time excellent performance and good portability.

1 Introduction

ML is a complicated programming language. Its syntax is complex and its dynamic semantics is related to the semantics of lexical Lisps like Scheme (cf [IEE90]). Furthermore it offers a strong polymorphic type system and a safe module system. Because of its overall qualities this language has become more and more popular for research and teaching, but with the emergence of better compilers it could reach a larger audience in industry as well.

At this time there are a few good implementations of ML, but all suffer at least from one or another drawback: disappointing execution speed, lack of portability or excessive use of memory.

The existence of a static type system in ML should permit us to compile this language nearly as efficiently as Pascal (not quite because polymorphism requires all objects to be the same size). The choice of C [KR88] as target language allows us to get a portable compiler easily because C is so widespread and now well standardized; nevertheless this choice must not cause bad general performance. Using the K2 compilation kit of Nitsan Séniak and Vincent Delacour, we built a very efficient and machine-independent ML to C compiler [Cri91].

In this article we explain our approach to ML compilation, provide a sketch of our implementation and give some results. We use the word 'global' to mean imported or exported and its contrary 'local' to mean local within a given module or function.

*To appear in the ACM SIGPLAN '92 Workshop on ML and its applications, San Francisco, June 20-21, 1992.

†Address: Laboratoire d'Informatique de l'École Normale Supérieure - 45 rue d'Ulm, 75230 Paris Cedex 05, France. Electronic mail: Regis.Cridlig@ens.fr

2 Compiling strategy

We use a new compilation technique, named *explicit specialization* [Sén89], to obtain very good optimization of function calls: the cost of function calls is indeed the bottleneck inherent to the compilation of all functional languages.

Our compiler reduces the semantic complexity of ML programs, according to some simple and well-defined steps, to a level acceptable by a compiler with a familiar technology like a typical C one. As a matter of fact people are accustomed to writing programs that use mostly the capabilities of classical algorithmic languages such as Pascal. In this case, the dynamic usage of local functions is limited to the static scope of their definitions, which fortunately allows us to handle functional values without allocating a closure in the heap: well-known and efficient techniques (cf [ASU86]) can be used to compile classical algorithmic languages.

Since data is handled in the obvious manner in our translation scheme and is later optimized sufficiently by modern C compilers, we concentrate our efforts and description on control flow.

2.1 The intermediate language: Sqil

The specialization technique consists of translating ML programs that use general control flow constructs into more specialized and better compilable forms, the Sqil ones.

The Sqil dialect, kernel of the K2 compilation kit, is a kind of lexical and bivalued[1] Lisp with continuations and functional values restricted to the global environment. Since Sqil only addresses the efficient compilation of control flow, it is limited to a few elementary special forms:

- `defun`, `defvar`, `progn`, `let`, `if`, `setq`, `labels` and `flet` are present with the same meaning as in Common Lisp (cf [Ste84]);
- `(function` *symbol*`)` yields the functional value of *symbol*, which must be a global function. This value can be later used in a computed call with `funcall`;
- `(the-continuation)` yields the calling continuation of the global current function definition. This continuation can be later invoked by `(continue` *cont value*`)`;
- finally, `(block` *name expressions...*`)` and `(return-from` *name value*`)` build lexical escapes.

The restriction on functional values and continuations guarantees one important property: identity between variables and registers (see [Sén91]), meaning that each local variable of a program can be allocated to a register. Additionally function call is direct by default thanks to bivaluation; computed calls explicitly use `funcall`.

We successfully used the K2 compiler written by Nitsan Séniak, who has developed original techniques to compile local functions towards C [Sén90]:

- *Displacement of continuations* propagates calling continuations of functions into the called functions in order to transform many non-tail calls into tail ones, which can be compiled using `goto`.

[1] I.e. symbols can have a functional value and a non-functional one at the same time, like in Common Lisp.

- *Function integration* in another function definition allows K2 to compile mutually tail calls between several functions into ordinary jumps.

The K2 compiler has proved to yield very good C code. Nevertheless Sqil could be directly compiled to efficient native code as well.

2.2 C as a target language

The Common Lisp compiler AKCL [Sch88] shows that it is realistic to compile Lisp or ML towards C used as a portable target language. Actually the primary advantages of C are:

- C is a low-level language, whose compilation captures all previously performed high-level optimizations well.
- C is highly portable and available on an increasing number of architectures, partly thanks to the spread of the Unix system.
- Thirdly, C compilers progress each year, while good assembly code generation by hand for RISC architectures is becoming very complicated.

However, in some respects, C is not so well-suited as a target language. Here are some reasons:

- The compiler designer loses full knowledge and mastery about the generated machine code: for instance, some critical optimizations concerning the function calling sequence that are performed by some Lisp compilers cannot be carried out any more. Moreover multiple results can be inefficiently compiled.
- Some specialized machine code instructions cannot be generated by the C compiler, like arithmetic operations with overflow checks. But this is less relevant for RISC architectures.
- Moreover, the arrangement of local variables in the C stack or in registers is left undefined. We cannot therefore use a classical garbage collector because it requires us to identify precisely all roots that can point to allocated data. Consequently we shall use a more complicated conservative garbage collector with ambiguous roots.

3 Implementation

3.1 The front-end

Our front-end is much the same as ZINC's one, written by Xavier Leroy [Ler90]. It first parses the Caml-Light grammar and then infers the types of the expressions using a variant of Milner's algorithm [Mil78]. The typechecker stores type information about each symbol in the global environment, which enables us to use the type of global functions during the code generation pass.

3.2 Code generation

3.2.1 Pattern-matching compilation

The pattern-matching compilation pass translates ML pattern-matching into a tree of Sqil's `case` selection instructions that is later translated to `switch` in C. The inherent backtracking involves the use of Sqil's lexical escape blocks, which generally compile to `goto`. This translation exactly follows the algorithm given in [Pey87], which is easy to implement but not optimal.

3.2.2 From ML abstract syntax tree to Sqil

We need two local environments for each toplevel phrase in this translation step:

1. An environment containing each variable with its access path from root to node in its defining pattern.
2. A local function environment containing the arity of each function name.

According to the different top nodes of the abstract syntax tree the translation proceeds in various ways:

- For the case of an identifier, we must distinguish a local or a global identifier, referring to a variable or to a function, because Sqil discriminates both kinds of values.
- For the case of a type constructor, we try to propagate constants, whenever it is possible thanks to the immutability of ML data types. Unfortunately this cannot be done with vectors and character strings that are inherently mutable in Caml-Light.
- The application case is the most complicated, because we want to uncurry most function calls:
  - When applying a local function or a primitive to a number of arguments matching its arity[2], we translate it to a direct application. But if some arguments are missing, we must add the code to build a partial function (using Sqil's `function` form); and if there are too many arguments, we have to compute additional calls using `funcall`.
  - The handling of global functions differs because the actual arity of an imported function is not accessible: only its type is known, and gives us its maximal arity. In order to increase execution speed, we insert several entry points for each global function, ranging from one argument to the maximum arity deducible from the function type. Thus a global function application simply calls the entry point corresponding to the number of available arguments.
  - If the functional value of the applicand is to be computed at runtime, we iterate `funcall` on each argument, since all functional values, i.e. closures, are of arity one in our scheme.

[2] We define the *arity* as the number of arguments available when the evaluation of the function can begin.

- The specialization of `let` needs the pattern-matching compilation pass and then distinguishes between variables, functions and recursive functions to be able to use `let`, `flet` and `labels` Sqil bivaluated forms.
- The CAML `try ... with` exception block is compiled by storing in a global variable the corresponding continuation, after having saved the old one. This permits us to raise an exception dynamically just by invoking the last created continuation.
- The translation of other constructs such as conditional forms or sequential forms is straightforward.

Finally we must remember to translate global definitions of functions and variables into `defun` and `defvar` Sqil forms.

3.2.3 The globalization pass

Up to this point we have not paid much attention to the fact that continuations and function values of Sqil are restricted to the toplevel environment. That ensures that these are simple adresses of either a stack frame or a global function entry.

If this is not the case, we have to globalize the function: a local function f that is referenced as a functional value must be moved to the toplevel and defined there with an additional argument representing its environment, i.e. its free variables. Then in general it is necessary to build a closure each time we need the functional value of f, and to send its current environment together with its argument whenever f is called.

Moreover each free function of f will be either integrated in f if it is only referenced in f, or else must be globalized too. So any free variables of these functions must be present in f's environment too.

If a local function contains the form `(the-continuation)`, it must be globalized as well, and additional arguments must be supplied for its free variables.

3.2.4 C code expansion and linking

When compiling Sqil code to C, remember that such optimizations as elimination of tail-recursive calls, continuation displacement and integration of local functions are performed. Macros for data handling are expanded too.

The Caml-Light module system is simple but quite natural and is based on interface files that provide the names and types of all exported global identifiers of a module. It allows separate compilation between modules and the definition of abstract types such as polymorphic hashtables. So we have to link all compiled modules (along with the runtime support) to obtain the executable file.

The standard library is entirely written in ML; it provides C written primitives only for low-level tasks such as I/O or Unix system calls.

3.3 The runtime system

3.3.1 The selected memory model

All ML objects are uniformly represented by a machine word. Characters, integers, short reals and some special sum types such as booleans are immediate values whereas

all other objects are statically or dynamically allocated in several words and represented by a pointer. We discuss below whether pointers and immediate values need to be distinguishable.

Closures and environments are simply allocated as a vector in consecutive memory words. So extending an environment by new variables is not implemented by chaining, but needs to copy the values into a bigger environment vector. Sum types require a small integer tag for the purpose of discriminating different constructors.

3.3.2 Which is the best garbage collector?

In order to collect memory which is no longer used, we need a garbage collector that can trace ambiguous roots in the C execution stack. It is possible to use a *mark and sweep* algorithm in a BIBOP memory model without any tags in the objects themselves. With this scheme *all* fields in an allocated object are ambiguous.

We have tried this scheme and used Boehm's portable collector [BW88] for a while, but the results were not very good for programs that allocate a great amount of data in the stack or in the heap. The garbage collector was indeed spending too much time in its mark function because it considers every word in the stack or in the heap as a potential pointer.

We think it is much better to distinguish pointers and immediate values with a tag. First the difference between, say, an integer and a pointer in the heap and even in the stack is much more immediate. Then one can use a mostly copying algorithm, such as Delacour's one [Del91], which is more efficient because it does not scan the whole heap like a *mark and sweep* algorithm and because it increases locality by copying and compacting the structured objects.

Moreover the runtime system can easily offer the user 'generic' functions such as `equal` that need to explore the tree structure of the objects, and give these functions a polymorphic type.

Our implementation uses a one-bit tag scheme that renders all immediate values even and all pointers odd. This tag scheme gives better results than the opposite one, since arithmetic operations are little altered while pointer dereference only needs a small offset, which is free on many machines.

3.4 Benchmarks

Some benchmarks are listed on figure 1. They were performed on a Decstation 3100 with the R2000 RISC processor. Tests on a Sparc architecture yield similar results.

We have chosen three other good compilers to make some comparisons. These are the well-known SML-NJ native code compiler [AM87], the fast bytecode interpreter Caml-Light and another ML to C compiler, Emmanuel Chailloux's CeML [Cha91]. Roughly speaking, SML-NJ uses a Continuation Passing Style translation scheme with a fast *stop and copy* generational garbage collector, and allocates everything on the heap [App87]. Caml-Light uses a similar runtime as ours, but its collector is a non-conservative one with generations, whereas CeML has a good conservative *mark and sweep* algorithm, but scans a special-purpose application stack instead of the C stack and uses a type tag in the objects themselves.

Compiler	SML-NJ	Caml-Light	CeML	Zinc→K2	
Version	0.75	0.41		without GC	with GC
Sieve	4.2	11.6	2.4	1.1	1.2
Solitaire	21.3	124.6	7.4	4.7	4.7
Takeushi	18.7	49.6	3.5	6.1	6.1
Tak w. exceptions	33.0	73.6	77	45.2	45.8
Knuth-Bendix	1.9	8.8	12	2.7	3.1
Boyer	5.6	22.7	n.a.	3.9	4.1
Euclidian division	3.4	20.2	17.5	2.7	3.1
Church integers	5.5	31.3	25.2	3.8	3.8
List summation	24.8	41.4	10.6	6.1	8.3
Integral	34.1	59.7	14.2	5.5	13.3

Figure 1: Comparison among some ML compilers (user times in seconds)

The first tests – Erathostenes' Sieve, the game of Solitaire and Takeushi's function – show that the C generated code is at its best when compiling imperative programs. That was indeed expected! The next two – Tak using exceptions to return each partial result and the Knuth-Bendix completion algorithm – demonstrate that exception handling is quite expensive in C, because you generally have to use the library functions `setjmp` and `longjmp`.

Knuth-Bendix completion procedure, Boyer's tautology checker, Euclidian division (extracted from the Coq proof assistant) and Church integers are very functional programs. Zinc→K2's excellent performances with respect to these programs show that with our optimizing compilation strategy C code generation can achieve the best performance results for typical ML programs.

List summation tests list processing whereas Integral tests floating point arithmetic. Except for this last test, garbage collection is cheap (actually the version without a GC uses a simple allocation fringe pointer and objects have no tag). The reason for this overhead is that we cannot tag short reals, so we have to box them. An alternative would be to adopt a more clever boxing mechanism (see [Ler92]).

4 Conclusions

We have demonstrated that C code generation can be very efficient if one succeeds in making a good use of the control flow instructions present in C, despite its bad exception handling mechanism. Uncurrying is a critical factor in achieving this, because it eliminates a lot of redundant partial applications. The use of an appropriate intermediate language permits us to describe the relevant optimizations easily.

The next step would consist of giving an object its natural C type whenever it is known statically to have a monomorphic ML type. This would allow the C compiler to handle the object directly, yielding a better assembly code, and would avoid many useless heap allocations. We plan to develop this idea and to add pertinent datatypes to Sqil.

Finally, a garbage collector with ambiguous roots can be (nearly) as efficient as a

conventional one, and a clever runtime tag mechanism does not hamper the overall performance.

5 Acknowledgements and related works

We would like to thank Xavier Leroy for his precious help, Emmanuel Chailloux for some valuable discussions, and Bruno Monsuez and Alan Mycroft respectively for their remarks about the first and second draft of this paper.

Previous work on the compilation of Lisp to C include the Kyoto Common Lisp [YH88] and Bartlett's SCHEME→C compiler [Bar89]. A previous attempt to compile ML to C is reported in [TAL90], but SML2C yields lower-level C programs, for example it does use an apply-like procedure for function calls instead of the standard C calling mechanism.

References

[AM87] A. Appel and D. MacQueen. A Standard ML compiler. In G. Kahn, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture*. Springer-Verlag, September 1987. LNCS Vol. 274.

[App87] A. Appel. Garbage collection can be faster than stack allocation. *Informations Processing Letters*, June 1987.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bar89] J. F. Bartlett. SCHEME→C: a portable Scheme-to-C compiler. Technical report, Digital Equipment Corporation, January 1989.

[BW88] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[Cha91] E. Chailloux. Compilation des langages fonctionnels : CeML un traducteur ML vers C. Doctorat de l'Université Paris VII, November 1991.

[Cri91] R. Cridlig. Compilateur optimisant pour le langage ML. Technical report, École Polytechnique, Palaiseau, France, July 1991.

[Del91] V. Delacour. Gestion mémoire automatique pour langages de programmation de haut niveau. Doctorat de l'Université Paris VI, Juin 1991.

[IEE90] IEEE standard for the Scheme programming language. Technical Report 1178, IEEE Std, 1990.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Software Series. Prentice Hall, 1988.

[Ler90] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[Ler92] X. Leroy. Unboxed objects and polymorphic typing. *POPL*, 1992.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[Pey87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Series in Computer Science. Prentice-Hall International, 1987.

[Sch88] W. Schelter. AKCL: Austin Kyoto Common Lisp. Unpublished(?), 1988.

[Sén89] N. Séniak. Compilation de Scheme par spécialisation explicite. *Bigre*, 1(65), July 1989.

[Sén90] N. Séniak. Efficient compilation of local functions using C as a back-end. Technical report, LIX, École Polytechnique, Palaiseau, France, 1990.

[Sén91] N. Séniak. Théorie et pratique de Sqil, un langage intermédiaire pour la compilation des langages fonctionnels. Doctorat de l'Université Paris VI, October 1991.

[Ste84] G. L. Steele. *Common Lisp: the Language*. Digital Press, 1984.

[TAL90] D. Tarditi, A. Acharya, and P. Lee. No assembly required: Compiling Standard ML to C. Technical Report 187, CMU-CS, November 1990.

[YH88] T. Yuasa and M. Hagiya. Kyoto Common Lisp Report. Technical Report, Kyoto University, Research Institute for Mathematical Sciences, 1988.