OCamlCC – Raising low-level bytecode to high-level C

Michel Mauny Benoît Vaugon
ENSTA-ParisTech
32, boulevard Victor,
F-75739 Paris Cedex 15, France
{Michel.Mauny,Benoit.Vaugon}@ensta.fr

Abstract

We present preliminary results about OCamlCC, a compiler producing native code from OCaml bytecode executables, through the generation of C code.

1. Introduction

The OCaml system [2] provides both a bytecode compiler, portable on virtually any architecture, and a native code generator that targets the most common ones.

Still, not being able to generate native code for less common architectures (too old, too new, or too rare) may be frustrating. The generation of native code for such uncommon architectures can be performed by translating OCaml programs to a language whose compilers are known to be efficient and widely available. The OCaml runtime being written in C, generating C code to be compiled with GCC looks reasonable.

At first sight, the idea is not new and there are quite a few compilers—Bigloo [5], SML2C [6], or GHC [4], to mention some of them—generating C from functional languages. However, our design has a few peculiarities that, to our knowledge, make it rather original. First of all, we want our executables to use the OCaml runtime (libraries, memory management), with as few modifications as possible. Ideally, we want to generate "high-level" C code: C functions with arguments, local variables, *etc.*, in order to take advantage of transformations and optimizations performed by the C compiler, and, ultimately, obtain reasonable performances.

2. What source language?

As already noticed by Vouillon and Balat in the design of their `js_of_ocaml` translator [8], OCaml bytecode executables provide a good starting point for such a translation: the design of the OCaml bytecode is stable, executables benefit from all the linking work performed by the OCaml compiler, and building tools processing such executables avoids modifying the OCaml compiler, simplifying greatly the maintenance and the evolution of such independent tools.

Generating and compiling C code from bytecode executables provides therefore a compilation route that extends bytecode generation, and can be used by application developers as a terminal step of the development process.

3. Generating C from bytecode

Roughly speaking, the execution of bytecode consists in interpreting code that essentially operates on an accumulator and a stack holding OCaml values. The stack, an array, stores arguments and intermediate values, and is also used by the garbage collector as the main root of the memory graph.

3.1 Bytecode macro-expansion

Generating (low-level) C from OCaml bytecode can be close to trivial: translate each bytecode instruction into a C macro that has been `#define`'d as the corresponding part of the bytecode interpreter. This is indeed the first implementation that we developed and, to say the truth, the starting point of our OCamlCC project.

Such a naive translation has one drawback: the resulting C program is essentially composed of *one* C function, whose body is a huge sequence of instructions. The resulting C file is large and the C compiler may need rather important resources (time, memory) when compiling big applications. For instance, generating C code from the OCaml compiler `ocamlc` produces a function whose body has more than 10^6 instructions and GCC just cannot compile it on machines with limited resources.

This macro-expansion has also good properties: terminal calls don't consume stack space, no change is needed in the OCaml runtime, and performances are encouraging: the generated code is always faster than the bytecode version but always slower than the native version (produced by `ocamlopt`). See figure 1 for some benchmarks, where column labeled `#define` shows execution times of macro-expanded programs compiled with `gcc -O3`. Figure 2 shows the sizes of the executables produced by the different compilers that we tested, and macro-expansion to C compiled with `gcc -O3` produces executables that are 1.6 to 2 times bigger than the executables produced by `ocamlopt`.

3.2 Decompilation

Low-level C code is not only bulky: the programming style (computed jumps, all values in a global array, . . .) is less amenable to optimization by the C compiler than a higher-level style, hence the need for a more clever translation.

The production of structured C from OCaml bytecode is done by isolating OCaml function bodies in the bytecode and generating one C function for each body. The prototype of the C functions is the following:

```
value f(value env);
```

where `value` is the C type of OCaml values and the `env` parameter denotes the environment part of the closures built from `f`.

Now that we have C functions corresponding to the OCaml abstractions in the original program, we optimize the code by:

- detecting static calls,
- performing "function pointer folding", that is, propagating closure components from their creation to where they are used,

	`ocamlc`	ratio	`#define + gcc -O3`	ratio	`ocamljit2`	ratio	`ocamlcc + gcc -O3`	`ocamlopt`	ratio
`almabench`	23.638	2.73	10.319	1.19	8.047	*0.93*	8.661	6.367	*0.74*
`almabench`^{unsafe}	23.308	2.72	10.284	1.20	6.522	*0.76*	8.564	6.335	*0.74*
`bdd`	7.319	8.21	1.073	1.20	1.868	2.10	0.891	0.427	*0.48*
`fft`	5.282	3.92	1.712	1.27	1.612	1.19	1.349	0.566	*0.42*
`fft`^{unsafe}	5.166	4.69	1.644	1.49	0.843	*0.76*	1.102	0.449	*0.41*
`fib`	10.883	20.15	2.658	4.92	3.778	7.00	0.540	0.875	1.62
`nucleic`	12.557	3.77	4.085	1.22	2.655	*0.80*	3.335	0.779	*0.23*
`quicksort`	5.225	9.06	0.893	1.55	1.382	2.40	0.577	0.217	*0.38*
`quicksort`^{unsafe}	5.034	10.67	0.706	1.50	0.936	1.98	0.472	0.179	*0.38*
`sorts`	18.748	4.98	4.879	1.30	5.615	1.49	3.764	2.546	*0.68*
`takc`	4.922	5.32	0.958	1.04	2.132	2.30	0.925	0.459	*0.50*
`taku`	6.652	4.17	2.176	1.36	2.708	1.70	1.597	0.434	*0.27*

Execution times, in seconds, on an Intel Xeon processor E5630 (12M cache, 2.53 GHz), running Fedora Linux 32 bits with GCC version 4.5.1.

The columns labeled "ratio" show the relative performance t/t_0 where t is the current execution time, and t_0 the time of the program compiled with `ocamlcc + gcc -O3`. A ratio greater that 1 indicates that `ocamlcc + gcc -O3` produces more efficient code than the compiler of the considered column.

Figure 1. Performances

	`ocamlc`	`#define + gcc -O3`	`ocamljit2`	`ocamlcc + gcc -O3`	`ocamlopt`
`almabench`	36K	348K	64K	340K	204K
`almabench`^{unsafe}	36K	348K	64K	340K	204K
`bdd`	8,0K	224K	20K	224K	132K
`fft`	8,0K	212K	20K	216K	128K
`fft`^{unsafe}	8,0K	212K	20K	216K	128K
`fib`	4,0K	200K	16K	196K	124K
`nucleic`	80K	444K	120K	416K	272K
`quicksort`	8,0K	208K	16K	204K	128K
`quicksort`^{unsafe}	8,0K	208K	16K	204K	128K
`sorts`	172K	972K	200K	944K	504K
`takc`	4,0K	200K	16K	196K	124K
`taku`	4,0K	200K	16K	196K	124K

Figure 2. The size of executables

- avoiding building unused closures,
- suppressing the environment parameter when unused,
- and extracting values (*e.g.* unallocated values) from the OCaml stack.

All these optimizations, excepted the last one, are classical and are performed using some kind of abstract interpretation techniques on the program[1].

The last optimization aims at transferring OCaml values from the OCaml stack to C local variables. We describe our technique in the next section.

4. From registered values to C variables

The bytecode interpreter stores all intermediate values in the stack of the bytecode interpreter, making them accessible to the garbage collector, and handles them indirectly because of the copying nature of the OCaml GC.

In our translation to C, it is frustrating to register unallocated values (integers, constant data constructors) in this stack whereas the GC will just ignore them. The goal of our optimization is to remove as many value registrations as possible and replace their indirect use through the stack by a direct handling as function arguments or local variables. The candidates to such a move are not only unallocated values. Precisely, a stack element shall be considered as a pointer (and hence cannot be extracted) if:

- it could have been stored as a pointer (there is at least one execution path that would store it as a new memory block, or as a value returned by an unknown function, ...),
- it could be read as a pointer (dereferenced as a block, passed to an unknown function, ...),
- a GC may occur during its "lifetime" (from the time it has been stored in the stack to the moment it has been popped off).

Stack elements that do not satisfy the conjunction of these three conditions can safely be extracted from the stack. This analysis is performed as a two-pass traversal of the bytecode.

When the stack element corresponds to a computed function argument, it will be passed directly to the corresponding C function, and the analysis is used to know whether it

[1] Since we analyze and translate a complete program, our static analyzes do not need to be modular.

must be registered in the stack as first action of the function. When the element is an intermediate result, the analysis decides whether it can be extracted to a C local variable. Of course, removing values from the stack, thus squeezing the stack, implies to re-adjust accesses from the code.

The advantages of moving values from the stack to variables are obviously that direct handling of values by C code is more efficient than indirect handling through the stack, but also that this allows the C compiler to perform more optimizations. Finally, having a smaller stack is also a gain since the garbage collector has slightly less work to perform.

Without this optimization, a partial application is managed by the function being called, just as the bytecode interpreter would do. Now that we have parameters to our C functions, we cannot call a function if some argument is missing. The callee is therefore unable to manage partial applications and we must transfer this task to the caller. The caller must know the arity of the callee: this is made available by adding this information as an extra field to closures, just as `ocamlopt` does.

We produced the bytecode of the concatenation of all benchmarks mentioned on figure 1 as a single file[2], and translated it to C with `ocamlcc -stat`, where the "`-stat`" option produces statistics about the code transformations mentioned above. The resulting code contains 2404 function calls, of which more than 70% have been detected as static and translated as such. Among the number of stack cell allocations[3], 44% were pushing statically known values and have therefore been suppressed. Among the 56% remaining stack cell allocations, 74% have been detected as storing values that could not be moved by the GC (see the conditions above) and have been translated as assignments to C local variables. In the end, on this example, less than 15% of stack cell allocations are left in the translated program.

5. Issues

Exceptions We use the classical `setjmp/longjmp` C primitives to encode OCaml exceptions, thus playing nicely with the exceptions from the OCaml runtime system, which use the same mechanism.

Tail calls It is well known that C compilers don't implement correctly tail calls, and this has been a problem for all compilers of functional languages using a C backend. We studied two solutions: one in pure C for portability and the other using assembly code for efficiency. We are currently comparing them but most probably, the former will be used as a fallback for architectures where the latter is not available.

Signals Signals cannot be checked at any time but only when memory is in a consistent state, since the control may be transferred to a piece of code that may trigger a garbage collection. The reception of a signal modifies a global flag that must periodically be checked for signals to be taken into account. Testing this flag has an impact on efficiency since it breaks the control flow, raising the question of when these checks are performed: at each allocation/loop/poptrap as does the bytecode compiler, or only at each allocation as does the native code compiler? Our current choice is, by default, to behave the same as the native code produced by `ocamlopt`, and to provide an option to our compiler for configuring signal management, allowing the programmer to choose between efficiency and reactivity.

Inlining the OCaml runtime In order to enable the inlining of library functions by the C compiler, we `#include` the OCaml runtime system in the generated C file. Doing so enables us to replace the startup functions and the callback mechanism by our own. However, such an inclusion generates name conflicts that we fixed by renaming the culprits (variables or macros).

C code size, resources needed for compiling We currently produce one C file, and, so far, compilation by GCC has never been a problem. For instance, compiling the C file generated for `ocamlc`, the OCaml bytecode compiler (with `gcc -O3`) takes a bit less than 2 minutes and uses less than 1GB of RAM. With `gcc -O0`, compilation time goes down to 34s and uses 610MB of RAM.

If compiling all-at-once were a problem, it would be easy to organize the generated C file in such a way that compiling it by fragments would be possible.

6. Performances

Figure 1 shows the execution times of programs compiled with the regular OCaml compilers (columns `ocamlc` and `ocamlopt`), with B. Meurer's OCamlJIT2 [3], and with our compiler in two versions: column `#define` is the naive translation described in section 3.1, and column `ocamlcc + gcc -O3` is our final translation, where OCaml values are extracted from the stack whenever possible.

The `fib` benchmark demonstrates the effectiveness of the extraction of values from the stack, making it more than 20 times faster than the bytecode interpreter and 1.6 times faster than `ocamlopt`'s native code.

Besides this somewhat artificial example, our code is from 2.7 to almost 10 times faster than the OCaml bytecode and `ocamlopt`'s native code is between 1.35 and 4.34 times faster than ours. The comparison between our code and OCamlJIT2 is generally favorable to OCamlCC excepted for benchmarks using intensively floating point arithmetic, which are optimized by OCamlJIT2 and not yet by OCamlCC. On benchmarks using integer and symbolic computations (`bdd`, `sorts`, *etc.*), we are from 1.49 to 2.4 times faster than OCamlJIT2.

7. Example

We use a very small example in order to show the C code produced by OCamlCC. The program that we compile is the following:

```
let rec fact n =
  if n<2 then 1 else n*fact(n−1) in
fact(10);;
```

We are here in a situation where the main function has no free variable (hence no need for a closure) and does not create or use allocated values (no need to go through the OCaml stack in its body). However, the generated bytecode is a little more complex than it may seem: a few global values need to be created and stored in a global table, some IO management (opening standard channels) is necessary, and the program terminates with a call to the *Pervasives.do_at_exit* function, that will, in turn, call *Pervasives.flush_all*.

The translation of this program first needs the bytecode executable `fact.byte`:

```
ocamlc fact.ml −o fact.byte
ocamlcc fact.byte −o fact.c
```

[2] This file contains therefore only one copy of the OCaml runtime and of the standard library.

[3] The number of push instructions contained in the program.

The file `fact.c` is almost 2500 lines long, and a large part of its content (*e.g.* functions from the OCaml standard library) will be unused and discarded by GCC. In order to obtain directly a small C file, one uses the OCamlClean program [7], that discards useless bytecode:

```
ocamlclean fact.byte
ocamlcc fact.byte −o fact.c
```

Now `fact.c` is 240 lines long, and has the following shape:

```
#include <ocamlcc.h> // Macros and the OCaml runtime system

/* OCaml global data */
intnat ocamlcc_global_data_length = 234;
char ocamlcc_global_data[] = {
  0x84, 0x95, 0xa6, 0xbe, 0x00, 0x00, 0x00, 0xd6,
   ...
  0x74, 0x40, 0x00
};

/* Functions for tail calls */
value ocamlcc_tail_call_p1;
value ocamlcc_tail_call_p2;
 ...

/* The Pervasives.do_at_exit function */
value f1(value p0, value env) {
  value v0;
  value *sp;
  sp = caml_extern_sp;
  sp[−1] = env;
  GETFIELD(0, Field(sp[−1], 2), v0);
  DYNAMIC_STANDARD_APPTERM(1, 1, Val_int(0), v0);
}

/* The (functional) value of exit_function
    inside Pervasives.at_exit */
value f6(value p0, value env) {
  value *sp;
  sp = caml_extern_sp;
  sp[−1] = env;
  DYNAMIC_APPLY(1, 1, 1, , Val_int(0), Field(sp[−1], 2));
  DYNAMIC_STANDARD_APPTERM(1, 1, Val_int(0), Field(sp[−1], 3));
}

/* The Pervasives.at_exit function */
value f14(value p0, value env) {
  value v0, v1;
  value tmp;
  value *sp;
  sp = caml_extern_sp;
  sp[−1] = env;
  sp[−2] = p0;
  GETFIELD(0, Field(sp[−1], 2), v0);
  MOVE(v0, sp[−3]);
  MAKE_YOUNG_BLOCK(4, 247, tmp, 3);
  SET_YOUNG_FIELD(0, tmp, (value) &f6);
  SET_YOUNG_FIELD(1, tmp, Val_int(1));
  SET_YOUNG_FIELD(2, tmp, sp[−2]);
  SET_YOUNG_FIELD(3, tmp, sp[−3]);
  MOVE(tmp, v0);
  MOVE(v0, v1);
  SETFIELD(0, Field(sp[−1], 2), v1);
  RETURN(Val_int(0));
}

/* The iter function, local to Pervasives.flush_all */
value f24(value p0, value env) {
  value v0, v1;
  value *sp;
  sp = caml_extern_sp;
  sp[−1] = env;
  sp[−2] = p0;
  BRANCHIFNOT(sp[−2], L00030);
  GETFIELD(1, sp[−2], v0);
  MOVE(v0, sp[−3]);
  GETFIELD(0, sp[−2], v0);
  MOVE(v0, sp[−4]);
  PUSHTRAP(, L00029, 96);
  MOVE(sp[−4], v0);
  CCALL(, caml_ml_flush, 4, v0);
  POPTRAP(4);
  BRANCH(L0002C);
 L00029:
 L0002C:
  MOVE(sp[−3], v0);
  MOVE(v0, v1);
  SPECIAL_SPECIAL_APPTERM(1, 2, 4, v1, Offset(sp[−1], 0));
 L00030:
  RETURN(Val_int(0));
}

/* The Pervasives.flush_all function */
value f46(value p0) {
  value v0, v1;
  value tmp;
  value *sp;
  sp = caml_extern_sp;
  MAKE_YOUNG_BLOCK(2, 247, tmp, 0);
  SET_YOUNG_FIELD(0, tmp, (value) &f24);
  SET_YOUNG_FIELD(1, tmp, Val_int(1));
  MOVE(Offset(tmp, 0), sp[−1]);
  MOVE(tmp, v0);
  CCALL(v0 =, caml_ml_out_channels_list, 1, Val_int(0));
  MOVE(v0, v1);
  MOVE(sp[−1], v0);
  STATIC_SPECIAL_APPTERM(2, f24, v1, v0);
}

/* Our factorial function */
value f75(value p0) {          // p0 is fact's parameter, no env.
  value v0, v1;                // local vars to hold intermediate results
  value *sp;
  sp = caml_extern_sp;
  BLEINT(2, p0, L00068);       // if (2≤p0) go to L00068
  RETURN(Val_int(1));          // otherwise return 1
 L00068:
  OFFSETINT(−1, p0, v0);       // v0 ← p0−1
  MOVE(v0, v1);                // v1 ← v0
  STATIC_APPLY(1, 0, 0, v0 =, f75, v1);     // v0 ← f75(v1)
  MOVE(v0, v1);                // v1 ← v0
  MULINT(p0, v1, v0);          // v0 ← p0 * v1
  RETURN(v0);                  // return v0
}

void ocamlcc_bytecode_main(void) {
  value v0;
  value tmp;
  value *sp;
  sp = caml_extern_sp;
  BRANCH(L0003D);
 L0003D:                  // initialization
  CCALL(, caml_ml_open_descriptor_in, 0, Val_int(0));
  CCALL(, caml_ml_open_descriptor_out, 0, Val_int(1));
  CCALL(, caml_ml_open_descriptor_out, 0, Val_int(2));
  MAKE_YOUNG_BLOCK(2, 247, tmp, 0);
  SET_YOUNG_FIELD(0, tmp, (value) &f46);
  SET_YOUNG_FIELD(1, tmp, Val_int(1));
  MOVE(tmp, v0);
  MAKE_SAVED_YOUNG_BLOCK(v0, 1, 0, tmp, 0);
  SET_YOUNG_FIELD(0, tmp, v0);
  MOVE(tmp, v0);
  MOVE(v0, sp[−1]);
  MOVE(sp[−1], v0);
```

```
    MAKE_SAVED_YOUNG_BLOCK(v0, 3, 247, tmp, 1);
    SET_YOUNG_FIELD(0, tmp, (value) &f1);
    SET_YOUNG_FIELD(1, tmp, Val_int(1));
    SET_YOUNG_FIELD(2, tmp, v0);
    MOVE(tmp, v0);
    MOVE(v0, sp[−2]);
    MOVE(v0, sp[−3]);
    CCALL(, caml_register_named_value, 2, Glob(12), sp[−3]);
    MOVE(sp[−2], v0);
    SETGLOBAL(13, v0);
    BRANCH(L00071);
   L00071:          // Call fact, a.k.a. f75, then do_at_exit()
    STATIC_APPLY(1, 0, 0, , f75, Val_int(10));
    STATIC_APPLY(1, 0, 0, , f1, Val_int(0), Glob(13));
    STOP;
  }
```

8. Future work

There remains much room for improving OCamlCC. We currently optimize function calls but we have so far paid no attention at improving other aspects of the execution of function bodies, which remain sequences of C macros corresponding to bytecode instructions. Many sub-sequences of those instructions (integer or floating point operations, *etc.*) need boxing and unboxing operations on their arguments and intermediate results. Some of those operations can be avoided, especially when we are sure that intermediate values remain "private" to the computation. In such cases, part of the computation can be optimized away and transferred to the corresponding C computations without using the OCaml boxing and unboxing primitives. OCamlJIT2 successfully does this on floating point operations and produces code that is competitive with `ocamlopt`'s optimized code [3].

More generally, since OCamlCC operates on a complete program, one could design and implement more aggressive interprocedural analyzes in order to further reduce boxing operations.

For compiling big applications on small machines, it will be necessary to divide the produced C code into chunks in order to separately compile them, and finally link the object files. This is only a matter of splitting the code into parts of reasonable size.

If we want OCamlCC to become a viable backend, we must also provide ways to do dynamic linking (generating and loading `.so` files).

C compilers such as GCC provide cross-compilation tools that should become accessible to OCaml programs through the use of OCamlCC. We currently have only little experience with cross-compilation and more experimentation is needed in order to exhibit and document cross-compilation for OCaml using OCamlCC.

On a longer term, other backends could be studied following a similar scheme. For instance, generating LLVM code out of the bytecode and linking it to the OCaml runtime system would probably provide an interesting experiment, to be compared to the LLVM backends currently under construction [1].

9. Conclusion

Through OCamlCC, we demonstrate that efficient binaries can be obtained by translating OCaml bytecode executables into C, providing an alternative backend to OCaml for architectures that are not supported by the OCaml native code generator.

The OCaml bytecode being one of the most stable parts of the OCaml system, the maintenance and development of OCamlCC is rather independent of the OCaml release schedule. OCamlCC can therefore live in harmony with the evolution of the OCaml system without needing to be in sync with the OCaml language or the internals of the OCaml compiler.

The idea of translating bytecode into a language that can be linked with the OCaml runtime system is also interesting in itself and should be explored further in targeting other languages.

References

[1] C. Benner. An LLVM Backend for OCaml. To be presented at the OCaml Meeting 2012.

[2] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system release 3.12, Documentation and user's manual, 2011. URL `http://caml.inria.fr/`.

[3] B. Meurer. OCamlJIT 2.0 - Faster Objective Caml. *CoRR*, abs/1011.1783, 2010.

[4] S. L. Peyton Jones, C. Hall, K. Hammond, J. Cordy, H. Kevin, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview, 1992.

[5] M. Serrano. Bigloo User's Manual. Technical Report RT-0169, INRIA, Dec. 1994. URL `http://hal.inria.fr/inria-00070001`. Projet ICSLA.

[6] D. Tarditi, P. Lee, and A. Acharya. No assembly required: compiling Standard ML to C. *ACM Lett. Program. Lang. Syst.*, 1(2):161–177, June 1992. ISSN 1057-4514. doi: 10.1145/151333.151343. URL `http://doi.acm.org/10.1145/151333.151343`.

[7] B. Vaugon. OCamlClean. Presented at the OCaml Meeting 2011.

[8] J. Vouillon and V. Balat. From bytecode to Javascript: the Js_of_ocaml compiler. Presented at the OCaml Meeting 2011.