27 The Compiler Backend: Bytecode and Native code

Once OCaml has passed the type checking stage, it can stop emitting syntax and type errors and begin the process of compiling the well-formed modules into executable code.

In this chapter, we'll cover the following topics:

- The untyped intermediate lambda code where pattern matching is optimized
- The bytecode `ocamlc` compiler and `ocamlrun` interpreter
- The native code `ocamlopt` code generator, and debugging and profiling native code

27.1 The Untyped Lambda Form

The first code generation phase eliminates all the static type information into a simpler intermediate *lambda form*. The lambda form discards higher-level constructs such as modules and objects and replaces them with simpler values such as records and function pointers. Pattern matches are also analyzed and compiled into highly optimized automata.

The lambda form is the key stage that discards the OCaml type information and maps the source code to the runtime memory model described in Chapter 24 (Memory Representation of Values). This stage also performs some optimizations, most notably converting pattern-match statements into more optimized but low-level statements.

27.1.1 Pattern Matching Optimization

The compiler dumps the lambda form in an s-expression syntax if you add the `-dlambda` directive to the command line. Let's use this to learn more about how the OCaml pattern-matching engine works by building three different pattern matches and comparing their lambda forms.

Let's start by creating a straightforward exhaustive pattern match using four normal variants:

```
type t = | Alice | Bob | Charlie | David

let test v =
  match v with
  | Alice   -> 100
  | Bob     -> 101
  | Charlie -> 102
  | David   -> 103
```

The lambda output for this code looks like this:

```
$ ocamlc -dlambda -c pattern_monomorphic_large.ml 2>&1
(setglobal Pattern_monomorphic_large!
  (let
    (test/86 =
       (function v/88 : int
         (switch* v/88
          case int 0: 100
          case int 1: 101
          case int 2: 102
          case int 3: 103)))
    (makeblock 0 test/86)))
```

It's not important to understand every detail of this internal form, and it is explicitly undocumented since it can change across compiler revisions. Despite these caveats, some interesting points emerge from reading it:

- There are no mentions of modules or types any more. Global values are created via `setglobal`, and OCaml values are constructed by `makeblock`. The blocks are the runtime values you should remember from Chapter 24 (Memory Representation of Values).
- The pattern match has turned into a switch case that jumps to the right case depending on the header tag of `v`. Recall that variants without parameters are stored in memory as integers in the order in which they appear. The pattern-matching engine knows this and has transformed the pattern into an efficient jump table.
- Values are addressed by a unique name that distinguishes shadowed values by appending a number (e.g., `v/1014`). The type safety checks in the earlier phase ensure that these low-level accesses never violate runtime memory safety, so this layer doesn't do any dynamic checks. Unwise use of unsafe features such as the `Obj.magic` module can still easily induce crashes at this level.

The compiler computes a jump table in order to handle all four cases. If we drop the number of variants to just two, then there's no need for the complexity of computing this table:

```
type t = | Alice | Bob

let test v =
  match v with
  | Alice   -> 100
  | Bob     -> 101
```

The lambda output for this code is now quite different:

```
$ ocamlc -dlambda -c pattern_monomorphic_small.ml 2>&1
(setglobal Pattern_monomorphic_small!
  (let (test/84 = (function v/86 : int (if v/86 101 100)))
    (makeblock 0 test/84)))
```

The compiler emits simpler conditional jumps rather than setting up a jump table, since it statically determines that the range of possible variants is small enough. Finally, let's consider code that's essentially the same as our first pattern match example, but with polymorphic variants instead of normal variants:

```
let test v =
  match v with
  | `Alice   -> 100
  | `Bob     -> 101
  | `Charlie -> 102
  | `David   -> 103
```

The lambda form for this also reflects the runtime representation of polymorphic variants:

```
$ ocamlc -dlambda -c pattern_polymorphic.ml 2>&1
(setglobal Pattern_polymorphic!
  (let
    (test/81 =
       (function v/83 : int
         (if (>= v/83 482771474) (if (>= v/83 884917024) 100 102)
           (if (>= v/83 3306965) 101 103))))
    (makeblock 0 test/81)))
```

We mentioned in Chapter 7 (Variants) that pattern matching over polymorphic variants is slightly less efficient, and it should be clearer why this is the case now. Polymorphic variants have a runtime value that's calculated by hashing the variant name, and so the compiler can't use a jump table as it does for normal variants. Instead, it creates a decision tree that compares the hash values against the input variable in as few comparisons as possible.

Pattern matching is an important part of OCaml programming. You'll often encounter deeply nested pattern matches over complex data structures in real code. A good paper that describes the fundamental algorithms implemented in OCaml is "Optimizing pattern matching"[1] by Fabrice Le Fessant and Luc Maranget.

The paper describes the backtracking algorithm used in classical pattern matching compilation, and also several OCaml-specific optimizations, such as the use of exhaustiveness information and control flow optimizations via static exceptions. It's not essential that you understand all of this just to use pattern matching, of course, but it'll give you insight as to why pattern matching is such an efficient language construct in OCaml.

27.1.2 Benchmarking Pattern Matching

Let's benchmark these three pattern-matching techniques to quantify their runtime costs more accurately. The `Core_bench` module runs the tests thousands of times and also calculates statistical variance of the results. You'll need to `opam install core_bench` to get the library:

[1] http://dl.acm.org/citation.cfm?id=507641

```
open Core
open Core_bench

module Monomorphic = struct
  type t =
    | Alice
    | Bob
    | Charlie
    | David

  let bench () =
    let convert v =
      match v with
      | Alice -> 100
      | Bob -> 101
      | Charlie -> 102
      | David -> 103
    in
    List.iter
      ~f:(fun v -> ignore (convert v))
      [ Alice; Bob; Charlie; David ]
end

module Monomorphic_small = struct
  type t =
    | Alice
    | Bob

  let bench () =
    let convert v =
      match v with
      | Alice -> 100
      | Bob -> 101
    in
    List.iter
      ~f:(fun v -> ignore (convert v))
      [ Alice; Bob; Alice; Bob ]
end

module Polymorphic = struct
  type t =
    [ `Alice
    | `Bob
    | `Charlie
    | `David
    ]

  let bench () =
    let convert v =
      match v with
      | `Alice -> 100
      | `Bob -> 101
      | `Charlie -> 102
      | `David -> 103
    in
    List.iter
```

```
      ~f:(fun v -> ignore (convert v))
      [ `Alice; `Bob; `Alice; `Bob ]
end

let benchmarks =
  [ "Monomorphic large pattern", Monomorphic.bench
  ; "Monomorphic small pattern", Monomorphic_small.bench
  ; "Polymorphic large pattern", Polymorphic.bench
  ]

let () =
  List.map benchmarks ~f:(fun (name, test) ->
      Bench.Test.create ~name test)
  |> Bench.make_command
  |> Command.run
```

Building and executing this example will run for around 30 seconds by default, and you'll see the results summarized in a neat table:

```
$ dune exec -- ./bench_patterns.exe -ascii -quota 0.25
Estimated testing time 750ms (3 benchmarks x 250ms). Change using
    '-quota'.

  Name                         Time/Run   Percentage
 ---------------------------- ---------- ------------
  Monomorphic large pattern     6.54ns       67.89%
  Monomorphic small pattern     9.63ns      100.00%
  Polymorphic large pattern     9.63ns       99.97%
```

These results confirm the performance hypothesis that we obtained earlier by inspecting the lambda code. The shortest running time comes from the small conditional pattern match, and polymorphic variant pattern matching is the slowest. There isn't a hugely significant difference in these examples, but you can use the same techniques to peer into the innards of your own source code and narrow down any performance hotspots.

The lambda form is primarily a stepping stone to the bytecode executable format that we'll cover next. It's often easier to look at the textual output from this stage than to wade through the native assembly code from compiled executables.

27.2 Generating Portable Bytecode

After the lambda form has been generated, we are very close to having executable code. The OCaml toolchain branches into two separate compilers at this point. We'll describe the bytecode compiler first, which consists of two pieces:

`ocamlc` Compiles files into a bytecode that is a close mapping to the lambda form
`ocamlrun` A portable interpreter that executes the bytecode

The big advantage of using bytecode is simplicity, portability, and compilation speed. The mapping from the lambda form to bytecode is straightforward, and this results in predictable (but slow) execution speed.

The bytecode interpreter implements a stack-based virtual machine. The OCaml stack and an associated accumulator store values that consist of:

long Values that correspond to an OCaml `int` type
block Values that contain the block header and a memory address with the data fields that contain further OCaml values indexed by an integer
code offset Values that are relative to the starting code address

The interpreter virtual machine only has seven registers in total: - program counter, - stack, exception and argument pointers, - accumulator, - environment and global data.

You can display the bytecode instructions in textual form via `-dinstr`. Try this on one of our earlier pattern-matching examples:

```
$ ocamlc -dinstr pattern_monomorphic_small.ml 2>&1
    branch L2
L1: acc 0
    branchifnot L3
    const 101
    return 1
L3: const 100
    return 1
L2: closure L1, 0
    push
    acc 0
    makeblock 1, 0
    pop 1
    setglobal Pattern_monomorphic_small!
```

The preceding bytecode has been simplified from the lambda form into a set of simple instructions that are executed serially by the interpreter.

There are around 140 instructions in total, but most are just minor variants of commonly encountered operations (e.g., function application at a specific arity). You can find full details online[2].

Where Did the Bytecode Instruction Set Come From?

The bytecode interpreter is much slower than compiled native code, but is still remarkably performant for an interpreter without a JIT compiler. Its efficiency can be traced back to Xavier Leroy's ground-breaking work in 1990, "The ZINC experiment: An Economical Implementation of the ML Language".a

This paper laid the theoretical basis for the implementation of an instruction set for a strictly evaluated functional language such as OCaml. The bytecode interpreter in modern OCaml is still based on the ZINC model. The native code compiler uses a different model since it uses CPU registers for function calls instead of always passing arguments on the stack, as the bytecode interpreter does.

Understanding the reasoning behind the different implementations of the bytecode interpreter and the native compiler is a very useful exercise for any budding language hacker.

a http://hal.inria.fr/docs/00/07/00/49/PS/RT-0117.ps

[2] http://cadmium.x9c.fr/distrib/caml-instructions.pdf

27.2.1 Compiling and Linking Bytecode

The `ocamlc` command compiles individual `ml` files into bytecode files that have a `cmo` extension. The compiled bytecode files are matched with the associated `cmi` interface, which contains the type signature exported to other compilation units.

A typical OCaml library consists of multiple source files, and hence multiple `cmo` files that all need to be passed as command-line arguments to use the library from other code. The compiler can combine these multiple files into a more convenient single archive file by using the `-a` flag. Bytecode archives are denoted by the `cma` extension.

The individual objects in the library are linked as regular `cmo` files in the order specified when the library file was built. If an object file within the library isn't referenced elsewhere in the program, then it isn't included in the final binary unless the `-linkall` flag forces its inclusion. This behavior is analogous to how C handles object files and archives (`.o` and `.a`, respectively).

The bytecode files are then linked together with the OCaml standard library to produce an executable program. The order in which `.cmo` arguments are presented on the command line defines the order in which compilation units are initialized at runtime. Remember that OCaml has no single `main` function like C, so this link order is more important than in C programs.

27.2.2 Executing Bytecode

The bytecode runtime comprises three parts: the bytecode interpreter, GC, and a set of C functions that implement the primitive operations. The bytecode contains instructions to call these C functions when required.

The OCaml linker produces bytecode that targets the standard OCaml runtime by default, and so needs to know about any C functions that are referenced from other libraries that aren't loaded by default.

Information about these extra libraries can be specified while linking a bytecode archive:

```
$ ocamlc -a -o mylib.cma a.cmo b.cmo -dllib -lmylib
```

The `dllib` flag embeds the arguments in the archive file. Any subsequent packages linking this archive will also include the extra C linking directive. This in turn lets the interpreter dynamically load the external library symbols when it executes the bytecode.

You can also generate a complete standalone executable that bundles the `ocamlrun` interpreter with the bytecode in a single binary. This is known as a *custom runtime* mode and is built as follows:

```
$ ocamlc -a -o mylib.cma -custom a.cmo b.cmo -cclib -lmylib
```

The custom mode is the most similar mode to native code compilation, as both generate standalone executables. There are quite a few other options available for compiling bytecode (notably with shared libraries or building custom runtimes). Full details can be found in the OCaml[3].

Dune can build a self-contained bytecode executable if you specify the byte_complete mode in the executable rule. For example, this dune file will generate a prog.bc.exe target:

```
(executable
  (name prog)
  (modules prog)
  (modes byte byte_complete))
```

27.2.3 Embedding OCaml Bytecode in C

A consequence of using the bytecode compiler is that the final link phase must be performed by ocamlc. However, you might sometimes want to embed your OCaml code inside an existing C application. OCaml also supports this mode of operation via the -output-obj directive.

This mode causes ocamlc to output an object file containing the bytecode for the OCaml part of the program, as well as a caml_startup function. All of the OCaml modules are linked into this object file as bytecode, just as they would be for an executable.

This object file can then be linked with C code using the standard C compiler, needing only the bytecode runtime library (which is installed as libcamlrun.a). Creating an executable just requires you to link the runtime library with the bytecode object file. Here's an example to show how it all fits together.

Create two OCaml source files that contain a single print line:

```
let () = print_endline "hello embedded world 1"
```

```
let () = print_endline "hello embedded world 2"
```

Next, create a C file to be your main entry point:

```
#include <stdio.h>
#include <caml/alloc.h>
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/callback.h>

int
main (int argc, char **argv)
{
  printf("Before calling OCaml\n");
  fflush(stdout);
  caml_startup (argv);
  printf("After calling OCaml\n");
  return 0;
}
```

Now compile the OCaml files into a standalone object file:

[3] https://ocaml.org/manual/comp.html#s%3Acomp-options

```
$ rm -f embed_out.c
$ ocamlc -output-obj -o embed_out.o embed_me1.ml embed_me2.ml
```

After this point, you no longer need the OCaml compiler, as `embed_out.o` has all of the OCaml code compiled and linked into a single object file. Compile an output binary using `gcc` to test this out:

```
$ gcc -fPIC -Wall -I`ocamlc -where` -L`ocamlc -where` -ltermcap -lm
    -ldl \
  -o finalbc.native main.c embed_out.o -lcamlrun
$ ./finalbc.native
Before calling OCaml
hello embedded world 1
hello embedded world 2
After calling OCaml
```

You can inspect the commands that `ocamlc` is invoking by adding `-verbose` to the command line to help figure out the GCC command line if you get stuck. You can even obtain the C source code to the `-output-obj` result by specifying a `.c` output file extension instead of the `.o` we used earlier:

```
$ ocamlc -output-obj -o embed_out.c embed_me1.ml embed_me2.ml
```

Embedding OCaml code like this lets you write OCaml that interfaces with any environment that works with a C compiler. You can even cross back from the C code into OCaml by using the `Callback` module to register named entry points in the OCaml code. This is explained in detail in the interfacing with C[4] section of the OCaml manual.

27.3 Compiling Fast Native Code

The native code compiler is ultimately the tool that most production OCaml code goes through. It compiles the lambda form into fast native code executables, with cross-module inlining and additional optimization passes that the bytecode interpreter doesn't perform. Care is taken to ensure compatibility with the bytecode runtime, so the same code should run identically when compiled with either toolchain.

The `ocamlopt` command is the frontend to the native code compiler and has a very similar interface to `ocamlc`. It also accepts `ml` and `mli` files, but compiles them to:

- A `.o` file containing native object code
- A `.cmx` file containing extra information for linking and cross-module optimization
- A `.cmi` compiled interface file that is the same as the bytecode compiler

When the compiler links modules together into an executable, it uses the contents of the `cmx` files to perform cross-module inlining across compilation units. This can be a significant speedup for standard library functions that are frequently used outside of their module.

Collections of `.cmx` and `.o` files can also be linked into a `.cmxa` archive by passing

[4] https://ocaml.org/manual/intfc.html

the -a flag to the compiler. However, unlike the bytecode version, you must keep the individual cmx files in the compiler search path so that they are available for cross-module inlining. If you don't do this, the compilation will still succeed, but you will have missed out on an important optimization and have slower binaries.

27.3.1 Inspecting Assembly Output

The native code compiler generates assembly language that is then passed to the system assembler for compiling into object files. You can get ocamlopt to output the assembly by passing the -S flag to the compiler command line.

The assembly code is highly architecture-specific, so the following discussion assumes an Intel or AMD 64-bit platform. We've generated the example code using -inline 20 and -nodynlink since it's best to generate assembly code with the full optimizations that the compiler supports. Even though these optimizations make the code a bit harder to read, it will give you a more accurate picture of what executes on the CPU. Don't forget that you can use the lambda code from earlier to get a slightly higher-level picture of the code if you get lost in the more verbose assembly.

The Impact of Polymorphic Comparison

We warned you in Chapter 15 (Maps and Hash Tables) that using polymorphic comparison is both convenient and perilous. Let's look at precisely what the difference is at the assembly language level now.

First let's create a comparison function where we've explicitly annotated the types, so the compiler knows that only integers are being compared:

```
let cmp (a:int) (b:int) =
  if a > b then a else b
```

Now compile this into assembly and read the resulting compare_mono.S file.

```
$ ocamlopt -S compare_mono.ml
```

This file extension may be lowercase on some platforms such as Linux. If you've never seen assembly language before, then the contents may be rather scary. While you'll need to learn x86 assembly to fully understand it, we'll try to give you some basic instructions to spot patterns in this section. The excerpt of the implementation of the cmp function can be found below:

```
_camlCompare_mono__cmp_1008:
        .cfi_startproc
.L101:
        cmpq    %rbx, %rax
        jle     .L100
        ret
        .align  2
.L100:
        movq    %rbx, %rax
        ret
        .cfi_endproc
```

The `_camlCompare_mono__cmp_1008` is an assembly label that has been computed from the module name (`Compare_mono`) and the function name (`cmp_1008`). The numeric suffix for the function name comes straight from the lambda form (which you can inspect using `-dlambda`, but in this case isn't necessary).

The arguments to `cmp` are passed in the `%rbx` and `%rax` registers, and compared using the `jle` "jump if less than or equal" instruction. This requires both the arguments to be immediate integers to work. Now let's see what happens if our OCaml code omits the type annotations and is a polymorphic comparison instead:

```
let cmp a b =
  if a > b then a else b
```

Compiling this code with `-S` results in a significantly more complex assembly output for the same function:

```
_camlCompare_poly__cmp_1008:
        .cfi_startproc
        subq    $24, %rsp
        .cfi_adjust_cfa_offset  24
.L101:
        movq    %rax, 8(%rsp)
        movq    %rbx, 0(%rsp)
        movq    %rax, %rdi
        movq    %rbx, %rsi
        leaq    _caml_greaterthan(%rip), %rax
        call    _caml_c_call
.L102:
        leaq    _caml_young_ptr(%rip), %r11
        movq    (%r11), %r15
        cmpq    $1, %rax
        je      .L100
        movq    8(%rsp), %rax
        addq    $24, %rsp
        .cfi_adjust_cfa_offset  -24
        ret
        .cfi_adjust_cfa_offset  24
        .align  2
.L100:
        movq    0(%rsp), %rax
        addq    $24, %rsp
        .cfi_adjust_cfa_offset  -24
        ret
        .cfi_adjust_cfa_offset  24
        .cfi_endproc
```

The `.cfi` directives are assembler hints that contain Call Frame Information that lets the debugger provide more sensible backtraces, and they have no effect on runtime performance. Notice that the rest of the implementation is no longer a simple register comparison. Instead, the arguments are pushed on the stack (the `%rsp` register), and a C function call is invoked by placing a pointer to `caml_greaterthan` in `%rax` and jumping to `caml_c_call`.

OCaml on x86_64 architectures caches the location of the minor heap in the `%r15` register since it's so frequently referenced in OCaml functions. The minor heap pointer can also be changed by the C code that's being called (e.g., when it allocates OCaml values), and so `%r15` is restored after returning from the `caml_greaterthan` call. Finally, the return value of the comparison is popped from the stack and returned.

Benchmarking Polymorphic Comparison

You don't have to fully understand the intricacies of assembly language to see that this polymorphic comparison is much heavier than the simple monomorphic integer comparison from earlier. Let's confirm this hypothesis again by writing a quick `Core_bench` test with both functions:

```
open Core
open Core_bench

let polymorphic_compare () =
  let cmp a b = Stdlib.(if a > b then a else b) in
  for i = 0 to 1000 do
    ignore(cmp 0 i)
  done

let monomorphic_compare () =
  let cmp (a:int) (b:int) = Stdlib.(if a > b then a else b) in
  for i = 0 to 1000 do
    ignore(cmp 0 i)
  done

let tests =
  [ "Polymorphic comparison", polymorphic_compare;
    "Monomorphic comparison", monomorphic_compare ]

let () =
  List.map tests ~f:(fun (name,test) -> Bench.Test.create ~name test)
  |> Bench.make_command
  |> Command.run
```

Running this shows quite a significant runtime difference between the two:

```
$ dune exec -- ./bench_poly_and_mono.exe -ascii -quota 1
Estimated testing time 2s (2 benchmarks x 1s). Change using '-quota'.

  Name                        Time/Run   Percentage
 ------------------------ ------------ ------------
  Polymorphic comparison   4_050.20ns      100.00%
  Monomorphic comparison     471.75ns       11.65%
```

We see that the polymorphic comparison is close to 10 times slower! These results shouldn't be taken too seriously, as this is a very narrow test that, like all such microbenchmarks, isn't representative of more complex codebases. However, if you're building numerical code that runs many iterations in a tight inner loop, it's worth manually peering at the produced assembly code to see if you can hand-optimize it.

Accessing Stdlib Modules from Within Core

In the benchmark above comparing polymorphic and monomorphic comparison, you may have noticed that we prepended the comparison functions with `Stdlib`. This is because the Core module explicitly redefines the > and < and = operators to be specialized for operating over `int` types, as explained in Chapter 15.1.4 (The Polymorphic Comparator). You can always recover any of the OCaml standard library functions by accessing them through the `Stdlib` module, as we did in our benchmark.

27.3.2 Debugging Native Code Binaries

The native code compiler builds executables that can be debugged using conventional system debuggers such as GNU `gdb`. You need to compile your libraries with the `-g` option to add the debug information to the output, just as you need to with C compilers.

Extra debugging information is inserted into the output assembly when the library is compiled in debug mode. These include the CFI stubs you will have noticed in the profiling output earlier (`.cfi_start_proc` and `.cfi_end_proc` to delimit an OCaml function call, for example).

Understanding Name Mangling

So how do you refer to OCaml functions in an interactive debugger like `gdb`? The first thing you need to know is how OCaml function names compile down to symbol names in the compiled object files, a procedure generally called *name mangling*.

Each OCaml source file is compiled into a native object file that must export a unique set of symbols to comply with the C binary interface. This means that any OCaml values that may be used by another compilation unit need to be mapped onto a symbol name. This mapping has to account for OCaml language features such as nested modules, anonymous functions, and variable names that shadow one another.

The conversion follows some straightforward rules for named variables and functions:

- The symbol is prefixed by `caml` and the local module name, with dots replaced by underscores.
- This is followed by a double `__` suffix and the variable name.
- The variable name is also suffixed by a `_` and a number. This is the result of the lambda compilation, which replaces each variable name with a unique value within the module. You can determine this number by examining the `-dlambda` output from `ocamlopt`.

Anonymous functions are hard to predict without inspecting intermediate compiler output. If you need to debug them, it's usually easier to modify the source code to let-bind the anonymous function to a variable name.

Interactive Breakpoints with the GNU Debugger

Let's see name mangling in action with some interactive debugging using GNU `gdb`.

Let's write a mutually recursive function that selects alternating values from a list. This isn't tail-recursive, so our stack size will grow as we single-step through the execution:

```
open Core

let rec take =
  function
  |[] -> []
  |hd::tl -> hd :: (skip tl)
and skip =
  function
  |[] -> []
  |_::tl -> take tl

let () =
  take [1;2;3;4;5;6;7;8;9]
  |> List.map ~f:string_of_int
  |> String.concat ~sep:","
  |> print_endline
```

Compile and run this with debugging symbols. You should see the following output:

```
(executable
  (name      alternate_list)
  (libraries core))
```

```
$ dune build alternate_list.exe
$ ./_build/default/alternate_list.exe -ascii -quota 1
1,3,5,7,9
```

Now we can run this interactively within gdb:

```
$ gdb ./alternate_list.native
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
    <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
    copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/avsm/alternate_list.native...done.
(gdb)
```

The gdb prompt lets you enter debug directives. Let’s set the program to break just before the first call to take:

```
(gdb) break camlAlternate_list__take_69242
Breakpoint 1 at 0x5658d0: file alternate_list.ml, line 5.
```

We used the C symbol name by following the name mangling rules defined earlier. A convenient way to figure out the full name is by tab completion. Just type in a portion of the name and press the <tab> key to see a list of possible completions.

Once you’ve set the breakpoint, start the program executing:

```
(gdb) run
Starting program: /home/avsm/alternate_list.native
```

```
[Thread debugging using libthread_db enabled]
Using host libthread_db library
    "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, camlAlternate_list__take_69242 () at alternate_list.ml:5
4          function
```

The binary has run until the first `take` invocation and stopped, waiting for further instructions. GDB has lots of features, so let's continue the program and check the backtrace after a couple of recursions:

```
(gdb) cont
Continuing.

Breakpoint 1, camlAlternate_list__take_69242 () at alternate_list.ml:5
4          function
(gdb) cont
Continuing.

Breakpoint 1, camlAlternate_list__take_69242 () at alternate_list.ml:5
4          function
(gdb) bt
#0  camlAlternate_list__take_69242 () at alternate_list.ml:4
#1  0x00000000005658e7 in camlAlternate_list__take_69242 () at
    alternate_list.ml:6
#2  0x00000000005658e7 in camlAlternate_list__take_69242 () at
    alternate_list.ml:6
#3  0x00000000005659f7 in camlAlternate_list__entry () at
    alternate_list.ml:14
#4  0x0000000000560029 in caml_program ()
#5  0x000000000080984a in caml_start_program ()
#6  0x00000000008099a0 in ?? ()
#7  0x0000000000000000 in ?? ()
(gdb) clear camlAlternate_list__take_69242
Deleted breakpoint 1
(gdb) cont
Continuing.
1,3,5,7,9
[Inferior 1 (process 3546) exited normally]
```

The `cont` command resumes execution after a breakpoint has paused it, `bt` displays a stack backtrace, and `clear` deletes the breakpoint so the application can execute until completion. GDB has a host of other features we won't cover here, but you can view more guidelines via Mark Shinwell's talk on "Real-world debugging in OCaml."[5]

One very useful feature of OCaml native code is that C and OCaml share the same stack. This means that GDB backtraces can give you a combined view of what's going on in your program *and* runtime library. This includes any calls to C libraries or even callbacks into OCaml from the C layer if you're in an environment which embeds the OCaml runtime as a library.

[5] http://www.youtube.com/watch?v=NF2WpWnB-nk%3C

27.3.3 Profiling Native Code

The recording and analysis of where your application spends its execution time is known as *performance profiling*. OCaml native code binaries can be profiled just like any other C binary, by using the name mangling described earlier to map between OCaml variable names and the profiler output.

Most profiling tools benefit from having some instrumentation included in the binary. OCaml supports two such tools:

- GNU `gprof`, to measure execution time and call graphs
- The Perf[6] profiling framework in modern versions of Linux

Note that many other tools that operate on native binaries, such as Valgrind, will work just fine with OCaml as long as the program is linked with the `-g` flag to embed debugging symbols.

Gprof

`gprof` produces an execution profile of an OCaml program by recording a call graph of which functions call one another, and recording the time these calls take during the program execution.

Getting precise information out of `gprof` requires passing the `-p` flag to the native code compiler when compiling *and* linking the binary. This generates extra code that records profile information to a file called `gmon.out` when the program is executed. This profile information can then be examined using `gprof`.

Perf

Perf is a more modern alternative to `gprof` that doesn't require you to instrument the binary. Instead, it uses hardware counters and debug information within the binary to record information accurately.

Run Perf on a compiled binary to record information first. We'll use our write barrier benchmark from earlier, which measures memory allocation versus in-place modification:

```
$ perf record -g ./barrier_bench.native
Estimated testing time 20s (change using -quota SECS).

  Name        Time (ns)             Time 95ci   Percentage
  ----        ---------             ---------   ----------
  mutable     7_306_219   7_250_234-7_372_469        96.83
  immutable   7_545_126   7_537_837-7_551_193       100.00

[ perf record: Woken up 11 times to write data ]
[ perf record: Captured and wrote 2.722 MB perf.data (~118926
    samples) ]
perf record -g ./barrier.native
Estimated testing time 20s (change using -quota SECS).

  Name        Time (ns)             Time 95ci   Percentage
```

[6] https://perf.wiki.kernel.org/

```
  ----        ---------              ---------   ----------
  mutable     7_306_219   7_250_234-7_372_469        96.83
  immutable   7_545_126   7_537_837-7_551_193       100.00

[ perf record: Woken up 11 times to write data ]
[ perf record: Captured and wrote 2.722 MB perf.data (~118926
    samples) ]
```

When this completes, you can interactively explore the results:

```
$ perf report -g
+  48.86%  barrier.native  barrier.native  [.]
    camlBarrier__test_immutable_69282
+  30.22%  barrier.native  barrier.native  [.]
    camlBarrier__test_mutable_69279
+  20.22%  barrier.native  barrier.native  [.] caml_modify
```

This trace broadly reflects the results of the benchmark itself. The mutable benchmark consists of the combination of the call to `test_mutable` and the `caml_modify` write barrier function in the runtime. This adds up to slightly over half the execution time of the application.

Perf has a growing collection of other commands that let you archive these runs and compare them against each other. You can read more on the home page[7].

Using the Frame Pointer to Get More Accurate Traces

Although Perf doesn't require adding in explicit probes to the binary, it does need to understand how to unwind function calls so that the kernel can accurately record the function backtrace for every event. Since Linux 3.9 the kernel has had support for using DWARF debug information to parse the program stack, which is emitted when the `-g` flag is passed to the OCaml compiler. For even more accurate stack parsing, we need the compiler to fall back to using the same conventions as C for function calls. On 64-bit Intel systems, this means that a special register known as the *frame pointer* is used to record function call history. Using the frame pointer in this fashion means a slowdown (typically around 3-5%) since it's no longer available for general-purpose use.

OCaml thus makes the frame pointer an optional feature that can be used to improve the resolution of Perf traces. opam provides a compiler switch that compiles OCaml with the frame pointer activated:

```
$ opam switch create 4.13+fp ocaml-variants.4.13.1+options
    ocaml-option-fp
```

Using the frame pointer changes the OCaml calling convention, but opam takes care of recompiling all your libraries with the new interface.

[7] `http://perf.wiki.kernel.org`

27.3.4 Embedding Native Code in C

The native code compiler normally links a complete executable, but can also output a standalone native object file just as the bytecode compiler can. This object file has no further dependencies on OCaml except for the runtime library.

The native code runtime is a different library from the bytecode one, and is installed as `libasmrun.a` in the OCaml standard library directory.

Try this custom linking by using the same source files from the bytecode embedding example earlier in this chapter:

```
$ ocamlopt -output-obj -o embed_native.o embed_me1.ml embed_me2.ml
$ gcc -Wall -I `ocamlc -where` -o final.native embed_native.o main.c \
   -L `ocamlc -where` -lasmrun -ltermcap -lm -ldl
$ ./final.native
Before calling OCaml
hello embedded world 1
hello embedded world 2
After calling OCaml
```

The `embed_native.o` is a standalone object file that has no further references to OCaml code beyond the runtime library, just as with the bytecode runtime. Do remember that the link order of the libraries is significant in modern GNU toolchains (especially as used in Ubuntu 11.10 and later) that resolve symbols from left to right in a single pass.

Activating the Debug Runtime

Despite your best efforts, it is easy to introduce a bug into some components, such as C bindings, that causes heap invariants to be violated. OCaml includes a `libasmrund.a` variant of the runtime library which is compiled with extra debugging checks that perform extra memory integrity checks during every garbage collection cycle. Running these extra checks will abort the program nearer the point of corruption and help isolate the bug in the C code.

To use the debug library, just link your program with the `-runtime-variant d` flag:

```
$ ocamlopt -runtime-variant d -verbose -o hello.native hello.ml
+ as  -o 'hello.o' '/tmp/build_cd0b96_dune/camlasmd3c336.s'
+ as  -o '/tmp/build_cd0b96_dune/camlstartup9d55d0.o'
    '/tmp/build_cd0b96_dune/camlstartup2b2cd3.s'
+ gcc -O2 -fno-strict-aliasing -fwrapv -pthread -Wall
    -Wdeclaration-after-statement -fno-common
    -fexcess-precision=standard -fno-tree-vrp -ffunction-sections
    -Wl,-E  -o 'hello.native'
    '-L/home/yminsky/.opam/rwo-4.13.1/lib/ocaml'
    '/tmp/build_cd0b96_dune/camlstartup9d55d0.o'
    '/home/yminsky/.opam/rwo-4.13.1/lib/ocaml/std_exit.o' 'hello.o'
    '/home/yminsky/.opam/rwo-4.13.1/lib/ocaml/stdlib.a'
    '/home/yminsky/.opam/rwo-4.13.1/lib/ocaml/libasmrund.a' -lm -ldl
$ ./hello.native
### OCaml runtime: debug mode ###
Initial minor heap size: 256k words
Initial major heap size: 992k bytes
Initial space overhead: 120%
Initial max overhead: 500%
Initial heap increment: 15%
Initial allocation policy: 2
Initial smoothing window: 1
Hello OCaml World!
```

27.4 Summarizing the File Extensions

We've seen how the compiler uses intermediate files to store various stages of the compilation toolchain. Here's a cheat sheet of all them in one place.

- `.ml` are source files for compilation unit module implementations.
- `.mli` are source files for compilation unit module interfaces. If missing, generated from the `.ml` file.
- `.cmi` are compiled module interface from a corresponding `.mli` source file.
- `.cmo` are compiled bytecode object file of the module implementation.
- `.cma` are a library of bytecode object files packed into a single file.
- `.o` are C source files that have been compiled into native object files by the system `cc`.
- `.cmt` are the typed abstract syntax tree for module implementations.
- `.cmti` are the typed abstract syntax tree for module interfaces.
- `.annot` are old-style annotation file for displaying `typed`, superseded by `cmt` files.

The native code compiler also generates some additional files.

- `.o` are compiled native object files of the module implementation.
- `.cmx` contains extra information for linking and cross-module optimization of the object file.
- `.cmxa` and `.a` are libraries of `cmx` and `o` units, stored in the `cmxa` and `a` files respectively. These files are always needed together.
- `.S` or `.s` are the assembly language output if `-S` is specified.