10

Program Analysis Tools

Program analysis tools provide supplementary information to the programmer in addition to the feedback from the compiler and the linker. Some of these tools perform a static analysis, i.e. they look at the code (either as text or in the form of a syntax tree) and determine certain properties like interdependency of modules or uncaught exceptions. Other tools perform a dynamic analysis, i.e. they look at the flow of execution. Analysis tools are useful for determining the number of calls to certain functions, getting a trace of the flow of arguments, or determining the time spent in certain parts of the program. Some are interactive, like the tools for debugging. In this case program execution is modified to account for user interaction. It is then possible to set breakpoints, in order to look at values or to restart program execution with different arguments.

The Objective Caml distribution includes such tools. Some of them have rather unusual characteristics, mostly dealing with static typing. It is, in fact, this static typing that guarantees the absence of type errors during program execution and enables the compiler to produce efficient code with a small memory footprint. Typing information is partly lost for constructed Objective Caml values. This creates certain difficulties, e.g. the impossibility of showing the arguments of polymorphic functions.

Chapter Overview

This short chapter presents the program analysis tools in the Objective Caml distribution. The first chapter describes the `ocamldep` command, which finds the dependencies in a set of Objective Caml files that make up an application.

The second section deals with debugging tools including tracing the execution of functions and the `ocamldebug` debugger, running under Unix.

The third section takes a look at the profiler, which can be used to analyze the execution of a program with an eye towards its optimization.

Dependency Analysis

Dependency analysis of a set of implementation and interface files that make up an Objective Caml application pursues a double end. The first is to get a global view of the interdependencies between modules. The second is to use this information in order to recompile only the absolutely necessary files after modifications of certain files.

The `ocamldep` command takes a set of `.ml` and `.mli` files and outputs the dependencies between files in `Makefile`[1] format.

These dependencies originate from global declarations in other modules, either by using dot.notation (e.g. `M1.f`) or by opening a module (e.g. `open  M1`).

Suppose the following files exist:

`dp.ml` :

```
let print_vect v =
    for i = 0 to Array.length v do
      Printf.printf "%f " v.(i)
    done;
    print_newline();;
```

and `d1.ml` :

```
let init n e =
  let v = Array.create 4 3.14 in
    Dp.print_vect v;
    v;;
```

Given the name of these files, the `ocamldep` command will output the following dependencies:

```
$ ocamldep dp.ml d1.ml array.ml array.mli printf.ml printf.mli
dp.cmo: array.cmi printf.cmi
dp.cmx: array.cmx printf.cmx
d1.cmo: array.cmi dp.cmo
d1.cmx: array.cmx dp.cmx
array.cmo: array.cmi
array.cmx: array.cmi
printf.cmo: printf.cmi
printf.cmx: printf.cmi
```

1. `Makefile` files are used by the `make` command for the maintenance of a set of programs or files to keep everything up to date after modifications to some of them.

The dependencies are determined for both the bytecode and the native compiler. The output is to be read in the following manner: production of the file `dp.cmo` depends on the files `array.cmi` and `printf.cmi`. Files with the extension `.cmi` depend on files with the same name and extension `.mli`. And the same holds by analogy for `.ml` files with `.cmo` and `.cmx` files.

The object files of the distribution do not show up in the dependency lists. In fact, if `ocamldep` does not find the files `array.ml` and `printf.ml` in the current directory, it will find them in the library directory of the installation and produce the following output:

```
$ ocamldep dp.ml d1.ml
d1.cmo: dp.cmo
d1.cmx: dp.cmx
```

To give new file search paths to the `ocamldep` command, the `-I directory` option is used, which adds a directory to the list of include directories.

Debugging Tools

There are two *debugging* tools. The first is a *trace* mechanism that can be used on the global functions in the toplevel loop. The second tool is a *debugger* that is not used in the normal toplevel loop. After a first program run it is possible to go back to breakpoints, and to inspect values or to restart certain functions with different arguments. This second tool only runs under Unix, because it duplicates the running process via a `fork` (see page 582).

Trace

The *trace* of a function is the list of the values of its parameters together with its result in the course of a program run.

The trace commands are directives in the toplevel loop. They allow to trace a function, stop its trace or to stop all active traces. These three directives are shown in the table below.

#trace name	trace function name
#untrace name	stop tracing function name
#untrace_all	stop all traces

Here is a first example of the definition of a function `f`:

```
# let f x = x + 1;;
val f : int -> int = <fun>
# f 4;;
```

```
- : int = 5
```

Now we will trace this function, so that its arguments and its return value will be shown.

```
#  #trace f;;
f is now traced.
# f 4;;
f <-- 4
f --> 5
- : int = 5
```

Passing of the argument `4` to `f` is shown, then the function `f` calculates the desired value and the result is returned and also shown. The arguments of a function call are indicated by a left arrow and the return value by an arrow to the right.

Functions of Several Arguments

Functions of several arguments (or functions returning a closure) are also traceable. Each argument passed is shown. To distinguish the different closures, the number of arguments already passed to the closures is marked with a `*`. Let the function `verif_div` take 4 numbers (a, b, q, r) corresponding to the integer division: $a = bq + r$.

```
# let verif_div a b q r =
    a = b*q + r;;
val verif_div : int -> int -> int -> int -> bool = <fun>
# verif_div 11 5 2 1;;
- : bool = true
```

Its trace shows the passing of 4 arguments:

```
#  #trace verif_div;;
verif_div is now traced.
# verif_div 11 5 2 1;;
verif_div <-- 11
verif_div --> <fun>
verif_div* <-- 5
verif_div* --> <fun>
verif_div** <-- 2
verif_div** --> <fun>
verif_div*** <-- 1
verif_div*** --> true
- : bool = true
```

Recursive Functions

The trace gives valuable information about recursive functions, e.g. poor stopping criteria are easily detected.

Let the function `belongs_to` which tests whether an integer belongs to a list of integers be defined in the following manner:

```
# let rec belongs_to (e : int) l = match l with
      []  → false
   | t :: q → (e = t) || belongs_to e q ;;
val belongs_to : int -> int list -> bool = <fun>
# belongs_to 4 [3;5;7] ;;
- : bool = false
# belongs_to 4 [1; 2; 3; 4; 5; 6; 7; 8] ;;
- : bool = true
```

The trace of the function invocation `belongs_to 4 [3;5;7]` will show the four calls of this function and the results returned.

```
#  #trace belongs_to ;;
belongs_to is now traced.
# belongs_to 4 [3;5;7] ;;
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [3; 5; 7]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [5; 7]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [7]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- []
belongs_to* --> false
belongs_to* --> false
belongs_to* --> false
belongs_to* --> false
- : bool = false
```

At each call of the function `belongs_to` the argument `4` and the list to search in are passed as arguments. When the list becomes empty, the functions return `false` as a return value which is passed along to each waiting recursive invocation.

The following example shows the section of the list when the element searched for appears:

```
# belongs_to 4 [1; 2; 3; 4; 5; 6; 7; 8] ;;
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [1; 2; 3; 4; 5; 6; 7; 8]
belongs_to <-- 4
belongs_to --> <fun>
```

```
belongs_to* <-- [2; 3; 4; 5; 6; 7; 8]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [3; 4; 5; 6; 7; 8]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [4; 5; 6; 7; 8]
belongs_to* --> true
belongs_to* --> true
belongs_to* --> true
belongs_to* --> true
- : bool = true
```

As soon as 4 becomes head of the list, the functions return `true` which gets passed along to each waiting recursive invocation.

If the sequence of statements around `||` were changed, the function `belongs_to` would still return the right result but would always have to go over the complete list.

```
# let rec belongs_to (e : int) = function
      []  → false
    | t :: q →   belongs_to e q || (e = t) ;;
val belongs_to : int -> int list -> bool = <fun>
# #trace belongs_to ;;
belongs_to is now traced.
# belongs_to 3 [3;5;7] ;;
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- [3; 5; 7]
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- [5; 7]
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- [7]
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- []
belongs_to* --> false
belongs_to* --> false
belongs_to* --> false
belongs_to* --> true
- : bool = true
```

Even though 3 is the first element of the list, it is traversed completely. So, trace also provides a mechanism for the efficiency analysis of recursive functions.

Polymorphic Functions

The trace does not show the value corresponding to an argument of a parameterized type. If for example the function `belongs_to` can be written without an explicit type constraint:

```
# let rec belongs_to e l = match l with
      []  → false
   | t :: q → (e = t) || belongs_to e q ;;
val belongs_to : 'a -> 'a list -> bool = <fun>
```

The type of the function `belongs_to` is now polymorphic, and the trace does no longer show the value of its arguments but replaces them with the indication (`poly`).

```
#  #trace belongs_to ;;
belongs_to is now traced.
# belongs_to 3 [2;3;4] ;;
belongs_to <-- <poly>
belongs_to --> <fun>
belongs_to* <-- [<poly>; <poly>; <poly>]
belongs_to <-- <poly>
belongs_to --> <fun>
belongs_to* <-- [<poly>; <poly>]
belongs_to* --> true
belongs_to* --> true
- : bool = true
```

The Objective Caml toplevel loop can only show monomorphic types. Moreover, it only keeps the inferred types of global declarations. Therefore, after compilation of the expression `belongs_to 3 [2;3;4]`, the toplevel loop in fact no longer possesses any further type information about the function `belongs_to` apart form the type *'a -> 'a list -> bool*. The (monomorphic) types of `3` and `[2;3;4]` are lost, because the values do not keep any type information: this is static typing. This is the reason why the trace mechanism attributes the polymorphic types *'a* and *'a list* to the arguments of the function `belongs_to` and does not show their values.

It is this absence of typing information in values that entails the impossibility of constructing a generic `print` function of type *'a -> unit*.

Local Functions

Local functions cannot be traced for the same reasons as above, relating again to static typing. Only global type declarations are kept in the environment of the toplevel loop. Still the following programming style is common:

```
# let belongs_to e l =
    let rec bel_aux l = match l with
      []  → false
    | t :: q → (e = t) || (bel_aux q)
    in
      bel_aux l;;
val belongs_to : 'a -> 'a list -> bool = <fun>
```

The global function only calls on the local function, which does the interesting part of the work.

Notes on Tracing

Tracing is actually the only multi-platform debugging tool. Its two weaknesses are the absence of tracing information for local functions and the inability to show the value of polymorphic parameters. This strongly restricts its usage, mainly during the first steps with the language.

Debug

`ocamldebug`, is a *debugger* in the usual sense of the word. It permits step-by-step execution, the insertion of breakpoints and the inspection and modification of values in the environment.

Single-stepping a program presupposes the knowledge of what comprises a *program step*. In imperative programming this is an easy enough notion: a step corresponds (more or less) to a single instruction of the language. But this definition does not make much sense in functional programming; one instead speaks of program *events*. These are applications, entries to functions, pattern matching, a conditional, a loop, an element of a sequence, etc.

Warning This tool only runs under Unix.

Compiling with Debugging *Mode*

The `-g` compiler option produces a `.cmo` file that allows the generation of the necessary instructions for debugging. Only the bytecode compiler knows about this option. It is necessary to set this option during compilation of the files encompassing an application. Once the executable is produced, execution in *debug* mode can be accomplished with the following `ocamldebug` command:

```
ocamldebug [options] executable [arguments]
```

Take the following example file `fact.ml` which calculates the factorial function:

```
let fact n =
  let rec fact_aux p q n =
    if n = 0 then p
    else fact_aux (p+q) p (n-1)
  in
fact_aux 1 1 n;;
```

The main program in the file `main.ml` goes off on a long recursion after the call of `Fact.fact` on −1.

```
let x = ref 4;;
let go () =
  x := -1;
  Fact.fact !x;;
```

```
go();;
```

The two files are compiled with the `-g` option:

```
$ ocamlc -g -i -o fact.exe fact.ml main.ml
val fact : int -> int
val x : int ref
val go : unit -> int
```

Starting the Debugger

Once an executable is compiled with *debug* mode, it can be run in this mode.

```
$ ocamldebug fact.exe
        Objective Caml Debugger version 3.00

(ocd)
```

Execution Control

Execution control is done via program events. It is possible to go forward and backwards by n program events, or to go forward or backwards to the next breakpoint (or the nth breakpoint). A breakpoint can be set on a function or a program event. The choice of language element is shown by line and column number or the number of characters. This locality may be relative to a module.

In the example below, a breakpoint is set at the fourth line of module `Main`:

```
(ocd) step 0
Loading program... done.
Time : 0
Beginning of program.
(ocd)  break @ Main 4
Breakpoint 1 at 5028 : file Main, line 4 column 3
```

The initialisations of the module are done before the actual program. This is the reason the breakpoint at line 4 occurs only after 5028 instructions.

We go forward or backwards in the execution either by program elements or by breakpoints. `run` and `reverse` run the program just to the next breakpoint. The first in the direction of program execution, the second in the backwards direction. The `step` command advanced by 1 or n program elements, entering into functions, `next` steps over them. `backstep` and `previous` respectively do the same in the backwards direction. `finish` finally completes the current functions invocations, whereas `start` returns to the program element before the function invocation.

To continue our example, we go forward to the breakpoint and then execute three program instructions:

```
(ocd) run
Time : 6 - pc : 4964 - module Main
Breakpoint : 1
4   <|b|>Fact.fact !x;;
(ocd) step
Time : 7 - pc : 4860 - module Fact
2   <|b|>let rec fact_aux p q n =
(ocd) step
Time : 8 - pc : 4876 - module Fact
6 <|b|>fact_aux 1 1 n;;
(ocd) step
Time : 9 - pc : 4788 - module Fact
3     <|b|>if n = 0 then p
```

Inspection of Values

At a breakpoint, the values of variables in the activation record can be inspected. The `print` and `display` commands output the values associated with a variable according to the different depths.

We will print the value of `n`, then go back three steps to print the contents of `x`:

```
(ocd) print n
n : int = -1
(ocd) backstep 3
Time : 6 - pc : 4964 - module Main
Breakpoint : 1
4   <|b|>Fact.fact !x;;
(ocd) print x
x : int ref = {contents=-1}
```

Access to the fields of a record or via the index of an array is accepted by the printing commands.

```
(ocd) print x.contents
1 : int = -1
```

Execution Stack

The execution stack permits a visualization of the entanglement of function invocations. The `backtrace` or `bt` command shows the stack of calls. The `up` and `down` commands select the next or preceding activation record. Finally, the `frame` command gives a description of the current record.

Profiling

This tool allows measuring a variety of metrics concerning program execution, including how many times a particular function or control structure (including conditionals, pattern matchers and loops) are executed. The results are recorded in a file. By examining this information, you may be able to locate either algorithmic errors or crucial locations for optimization.

In order for the profiler to do its work, it is necessary to compile the code using a special mode that adds profiling instructions. There are two *profiling* modes: one for the bytecode compiler, and the other for the native-code compiler. There are also two commands used to analyze the results. Analysis of native code will retrieve the time spent in each function.

Profiling an application therefore proceeds in three stages:

1. compilation in *profiling* mode;
2. program execution;
3. presentation of measurements.

Compilation Commands

The commands to compile in *profiling* mode are the following:

- `ocamlcp -p` *options* for the bytecode compiler;
- `ocamlopt -p` *options* for the native-code compiler.

These compilers produce the same type of files as the usual commands (see chapter 7). The different options are described in figure 10.1.

`f`	function call
`i`	branch of **`if`**
`l`	**`while`** and **`for`** loops
`m`	branches of **`match`**
`t`	branches of **`try`**
`a`	all options

Figure 10.1: Options of the *profiling* commands

These indicate which control structures must be taken into account. By default, the `fm` options are activated.

Program Execution

Bytecode Compiler

The execution of a program compiled in profiling mode will, if it terminates, produce a file named `ocamlprof.dump` which contains the information wanted.

We resume the example of the product of a list of integers. We write the following file `f1.ml`:

```
let rec interval a b =
  if b < a then []
  else a :: (interval (a+1) b);;

exception Found_zero ;;

let mult_list l =
 let rec mult_rec l = match l with
     [] → 1
   | 0::_ → raise Found_zero
   | n :: x → n * (mult_rec x)
 in
  try mult_rec l with Found_zero → 0
;;
```

and the file `f2.ml` which uses the functions of `f1.ml`:

```
let l1 = F1.interval 1 30;;
let l2 = F1.interval 31 60;;
let l3 = l1 @ (0 :: l2);;

print_int (F1.mult_list l1);;
print_newline();;

print_int (F1.mult_list l3);;
print_newline();;
```

The compilation of these files in profiling mode is shown in the following:

```
ocamlcp -i -p a -c f1.ml
val profile_f1_ : int array
val interval : int -> int -> int list
exception Found_zero
val mult_list : int list -> int
```

With the `-p` option, the compiler adds a new function (`profile_f1_`) for the initialization of the counters in module `F1`. It is the same for file `f2.ml`:

```
ocamlcp -i -p a -o f2.exe f1.cmo f2.ml
val profile_f2_ : int array
val l1 : int list
val l2 : int list
val l3 : int list
```

Native Compiler

The native code compilation gives the following result:

```
$ ocamlopt -i -p  -c f1.ml
val interval : int -> int -> int list
exception Found_zero
val mult_list : int list -> int
$ ocamlopt -i -p -o f2nat.exe f1.cmx f2.ml
```

Only the `-p` option without argument is used. The execution of `f2nat.exe` produces a file named `gmon.out` which is in a format that can be handled by the usual Unix commands (see page 284).

Presentation of the Results

Since the information gathered by the two *profiling* modes differs, their presentation follows suit. In the first (bytecode) mode comments on the number of passages through the control structures are added to the program text. In the second (native) mode, the time spent in its body and the number of calls is associated with each function.

Bytecode Compiler

The `ocamlprof` command gives the analysis of the measurement results. It uses the information contained in the file `camlprof.dump`. This command takes the source of the program on entry, then reads the measurements file and produces a new program text with the desired counts added as comments.

For our example this gives:

```
ocamlprof f1.ml

let rec interval a b =
  (* 62 *) if b < a then (* 2 *) []
  else (* 60 *) a::(interval (a+1) b);;

exception Found_zero ;;

let mult_list l =
 (* 2 *) let rec mult_rec l = (* 62 *) match l with
```

```
    [] -> (* 1 *) 1
  | 0::_ -> (* 1 *) raise Found_zero
  | n::x -> (* 60 *) n * (mult_rec x)
 in
  try mult_rec l with Found_zero -> (* 1 *) 0
;;
```

These counters reflect the calculations done in F2 quite well. There are two calls of `mult_list` and 62 of the auxiliary function `mult_rec`. Examination of the different branches of the pattern matching show 60 passages through the common case, one through the pattern [] and the only match where the head is 0, raising an exception, which can be seen in the counter of the **try** statement.

The `ocamlprof` command accepts two options. The first *-f file* indicates the name of the file to contain the measurements. The second `-F string` specifies a string to add to the comments associated with the control structures treated.

Native Compilation

To get the time spent in the calls of the functions for multiplying the elements of a list, we write the following file `f3.ml`:

```
let l1 = F1.interval 1 30;;
let l2 = F1.interval 31 60;;
let l3 = l1 @ (0 :: l2);;

for i=0 to 100000 do
  F1.mult_list l1;
  F1.mult_list l3
done;;

print_int (F1.mult_list l1);;
print_newline();;

print_int (F1.mult_list l3);;
print_newline();;
```

This is the same file as `f2.ml` with a loop of 100000 iterations.

Execution of the program creates the file `gmon.out`. This is in a format readable by `gprof`, a command that can be found on Unix systems. The following call to `gprof` prints information about the time spent and the call graph. Since the output is rather long, we show only the first page which contains the name of the functions that are called at least once and the time spent in each.

```
$ gprof  f3nat.exe
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 92.31       0.36     0.36   200004      1.80      1.80  F1_mult_rec_45
  7.69       0.39     0.03   200004      0.15      1.95  F1_mult_list_43
  0.00       0.39     0.00     2690      0.00      0.00  oldify
  0.00       0.39     0.00      302      0.00      0.00  darken
  0.00       0.39     0.00      188      0.00      0.00  gc_message
  0.00       0.39     0.00      174      0.00      0.00  aligned_malloc
  0.00       0.39     0.00      173      0.00      0.00  alloc_shr
  0.00       0.39     0.00      173      0.00      0.00  fl_allocate
  0.00       0.39     0.00       34      0.00      0.00  caml_alloc3
  0.00       0.39     0.00       30      0.00      0.00  caml_call_gc
  0.00       0.39     0.00       30      0.00      0.00  garbage_collection
...
```

The main lesson is that almost all of the execution time is spent in the function `F1_mult_rec_45`, which corresponds to the function `F1.mult_rec` in file `f1.ml`. On the other hand we recognize a lot of other functions that are called. The first on the list are memory management functions in the runtime library (see chapter 9).

Exercises

Tracing Function Application

This exercise shows the evaluation of arguments at the moment of function application.

1. Activate tracing of the function `List.fold_left` and evaluate the following expression:
   `List.fold_left (-) 1 [2; 3; 4; 5];;`
   What does the trace show you?
2. Define the function `fold_left_int`, identical to `List.fold_left`, but with type:
   (int → int → int) → int → int list → int.
   Trace this function. Why is the output of the trace different?

Performance Analysis

We continue the exercise proposed in chapter 9, page 247, where we compared the evolution of the heap of two programs (one tail recursive and the other not) for calculating primes. This time we will compare the execution times of each function with the *profiling* tools. This exercise shows the importance of inline expansion (see chapter 7).

1. Compile the two programs `erart` and `eranrt` with *profiling* options using the bytecode compiler and the native code compiler respectively.
2. Execute the programs passing them the numbers 3000 4000 5000 6000 on the command line.
3. Visualize the results with the `ocamlprof` and `gprof` commands. What can you say about the results?

Summary

This chapter presented the different programming support tools that come with the Objective Caml distribution.

The first tool performs a static analysis in order to determine the dependencies of a set of compilation units. This information can then be put in a `Makefile`, allowing for separate compilation (if you alter one source file in a program, you only have to compile that file, and the files that have dependencies to it, rather than the entire program).

Other tools give information about the execution of a program. The interactive toplevel offers a trace of the execution; but, as we have seen, polymorphism imposes quite heavy restrictions on the observable values. In fact, only the global declarations of monomorphic values are visible, which nevertheless includes the arguments of monomorphic functions and permits tracing of recursive functions.

The last tools are those in the tradition of development under Unix, namely a *debugger* and a *profiler*. With the first, you can execute a program step by step to examine it's operation, and the second gives information about its performance. Both are usable only under Unix.

To Learn More

The results produced by the `ocamldep` command can be visualized in graphical form by the `ocamldot` utility, which can be found on the following page:

Link: http://www.cis.upenn.edu/~tjim/ocamldot/index.html

`ocamldot` makes use of an independent program (`dot`), also downloadable:

Link: http://www.research.att.com/sw/tools/graphviz/

Several generic `Makefile` templates for Objective Caml have been proposed to ease the burden of project management:

Link: http://caml.inria.fr/FAQ/Makefile_ocaml-eng.html

Link: http://www.ai.univie.ac.at/~markus/ocaml_sources

These integrate the output of `ocamldep`.

In [HF$^+$96] a performance evaluation of about twenty implementations of functional languages, among them several ML implementations, can be found. The benchmark is an example of numerical calculations on large datastructures.