Real-world debugging in OCaml

Mark Shinwell

Jane Street Europe

OCaml Users and Developers Workshop 2012

My program has a bug

- Everyday debugging: use `printf`.
  - Don't forget to flush: `Printf.printf "foo\n%!"`
- Know your standard Unix tools
  - I'm not sure which server it connects to

    ```
    strace foo_client.exe 2>&1 | grep connect
    ```
  - I want to know which files it has open

    ```
    lsof -p 12345
    ```
  - I need to check where it finds `input.txt`

    ```
    strings foo_client.exe | grep input.txt
    ```

JANE STREET

My bug resists attempts to find it

- Make basic checks on the machine
  - disk space, memory, errors in the system logs
- Ensure backtraces are enabled
  - `export OCAMLRUNPARAM=b`
- Turn on core dumps
  - `ulimit -c unlimited`
- Recompile your C stubs (and the OCaml runtime)
  - No optimization: `-O0`
  - With debugging info: `-g`

JANE STREET

My program needs a debugger

- `gdb` does work with OCaml programs
- Support is significantly improved in OCaml 4.00
  - backtraces
  - source file locations
- Names in the debugger are mangled
  - `camlList__iter_1074` $\equiv$ `List.iter`
- Printing and traversal of OCaml values can be tricky

JANE STREET

My gdb-foo is awful

- New program: `gdb --args myprogram.exe --foo --bar`
- Attach to running program: `gdb -p 14001`
- Useful commands:
  - `r` and `c` – run / continue running program
  - `b` – set breakpoint
  - [`thr apply all`] `bt` – backtrace [for all threads]
  - `inf thr` – state of all threads
  - `p` and `x` – examine values and memory
  - `step` and `next` – single stepping
  - `inf reg` – state of CPU registers
  - `Ctrl+C` and `q` – return to / exit from debugger

JANE STREET

My program needs to be stopped... just here

- Breakpoint conditions in `gdb` can be hard to use
  - the condition may be hard to express
  - decoding the OCaml values makes this doubly hard

- Send a stop signal to yourself and then attach `gdb`

```
let draw_shape ~x ~y = function
  | Square when x < 200 ->
    Low_level_debug.stop_me_now ()
  | ...
```

JANE STREET

My gdb backtrace is useless

- Dump the stack and look for code pointers:

```
(gdb) x/256x $rsp
...
7fffffffe3c0:  0000000a 00000000 0070d708 00000000
7fffffffe3d0:  000050b5 00000000 006b51c5 00000000
7fffffffe3e0:  00bf1338 00000000 ab515268 00002aaa
```

- Turn a code pointer into a function name using `objdump`:

```
00000000006b51a0 <camlList__iter_1074>:
...
6b51c5:  48 8b 04 24   mov (%rsp),%rax
```

JANE STREET

My program fails with an uncaught exception

- Perhaps it fails before it actually *does* anything
  - Top-level expression with a side effect?

- A backtrace may not be sufficient to find the bug
- Try to catch it before it exits:

```
(gdb) b caml_fatal_uncaught_exception
(gdb) r
...
(gdb) bt
```

My program exits at some random point

- Perhaps there is no exception visible, for whatever reason
- Set breakpoints:

```
(gdb) b exit
(gdb) b caml_sys_exit
```

- `gdb` can go back in time

`sourceware.org/gdb/wiki/ReverseDebug`

JANE STREET

My code is camoflaging the real exception

- With Core on x86-64: backtraces on demand

```
let f x =
  ...
  Printf.eprintf "f was called from: %s\n%!"
    (Backtrace.to_string (Backtrace.get ()));
  ...
```

- Can also be invoked from inside gdb

```
(gdb) call backtrace_dump_stderr()
```

JANE STREET

My program segfaults

```
kernel:  myprogram[14001]:  segfault at 00002aaaac001280
  rip 0000000000e000f8 rsp 7fffffffffdd0000 error 15
```

- The information is
  - the process name and ID
  - which address the program was trying to access
  - which instruction caused the fault
  - the stack pointer at the time of the fault
  - what was attempted (e.g. an instruction fetch)

My code shouldn't segfault!

- Stack overflow
  - backtrace may show excessive number of stack frames
  - increase stack limit: `ulimit -s`
- Corruption in the Caml heap
  - segfault often lies in the GC (e.g. `caml_oldify_one`)
  - usually caused by faulty C bindings
- Hardware failure
  - the instruction pointer may be way outside your code
  - check the system logs for excessive segfaults

JANE STREET

My program deadlocks

- Should be evident using "`info threads`" in `gdb`
  - One thread wants mutex B whilst holding mutex A
  - Another thread wants mutex A whilst holding mutex B

JANE STREET

My C bindings seem to be faulty

- Read them carefully
  - Every variable that is live across an allocation point
  - Every block where you release the runtime lock
  - If you use a `CAML...` macro, always use `CAMLreturn`
- Run the program under `valgrind`
  - Will not catch corruption within the Caml heap
- (Re-)write them carefully
  - Use `assert` to check values are what you think they are
  - Don't release the runtime lock unless you really have to.

JANE STREET

My C bindings still seem to be faulty

- Use of the GC registration macros at every possible opportunity does not guarantee correctness:

```
value works_most_of_the_time(value v_filename)
{
  CAMLparam1(v_filename);
  char* filename = String_val(v_filename);
  caml_enter_blocking_section();
  takes_a_long_time(filename);
  caml_leave_blocking_section();
  CAMLreturn(Val_unit);
}
```

Conclusion

- Standard tools can be used to debug OCaml
- OCaml 4 offers significant improvements

- Don’t forget: there’s a logical explanation for every bug

JANE STREET