Camlboot: debootstrapping the OCaml compiler

Nathanaëlle Courant, Julien Lepiller, Gabriel Scherer

March 7, 2022

What is debootstrapping?

- *Source file*: preferred form for human editing and understanding
- *Self-bootstrapping* a compiler: compiling it with itself
  $\Longrightarrow$ Need (non-source) binaries of the compiler to build the compiler
- *Debootstrapping* a compiler: building a compiler without using its self-bootstrapped binaries

Why debootstrap? (1/4)

Trusting trust attack: bugs (or malicious code) can reproduce themselves through bootstrap binaries:

- some bugs seen in the wild, rarely reported,
- proofs of concept in Rust and Go,
- Induc virus: reproduces itself through Delphi compilers, discovered in the the wild in 2009, fortunately harmless!

Countering trusting trust: diverse double compilation

Diverse Double Compilation (DDC): use an independent compiler B to check that a deterministic compiler A is free from trusting-trust attacks.

- Compile A with both A and B
  $\Longrightarrow$ different binaries, but semantics should be the same.
- Compile A with the resulting binaries
  $\Longrightarrow$ should get the same output.

Why debootstrap? (2/4)

License question: is software free if:

- you need a proprietary compiler to build it?
- you need a proprietary compiler to build its compiler?
- there is no way to build it without using binaries at some point?

Why debootstrap? (3/4)

Reproducible builds: bit-for-bit identical results for software built twice in the same environment, allows caching and verification.

- Can it be trusted if the environment already contains the output?

Why debootstrap? (4/4)

Semantics question: can we really specify the semantics of a program when some of it is hidden inside the compiler binary (and not source)?

```
let unescape_char c =
  match c with
  | 'n' -> '\n'
  | 't' -> '\t'
  [...]
```

How to debootstrap?

- Legacy path: replay compilation using a chain of old implementations
- Tailored path: use new implementations to shorten the chain

Key metric: total human work required

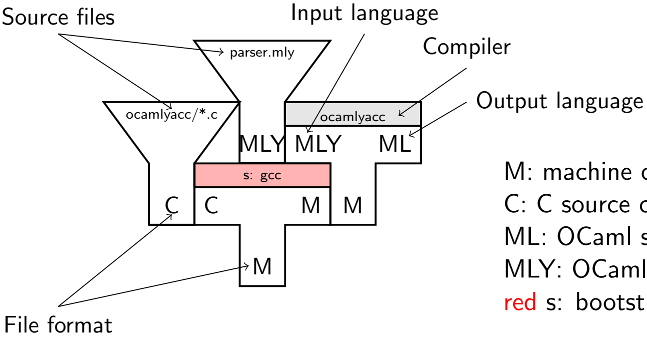
Writing a new implementation can be faster than finding and making old implementations work (also, much more interesting).

Components of Camlboot

- `interp`: An interpreter for almost all of OCaml, able to run the OCaml compiler
  - Written in MiniML, a subset of OCaml
  - Reuses the OCaml parser and lexer
- `minicomp`: A compiler from MiniML to OCaml bytecode
  - Written in Scheme
  - Very naïve
- A handwritten lexer to solve the bootstrap of `ocamllex`

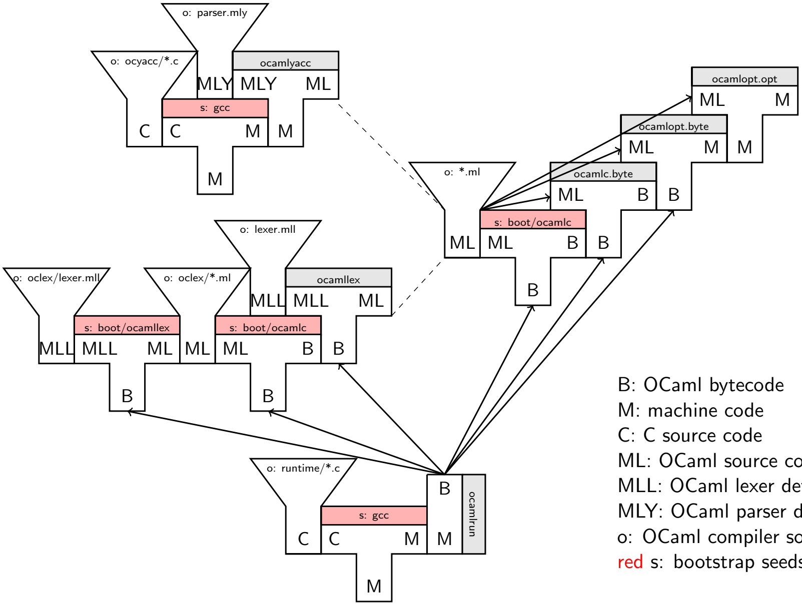
T-diagrams

T-diagram: graphical depiction of source file, output file, and compiler



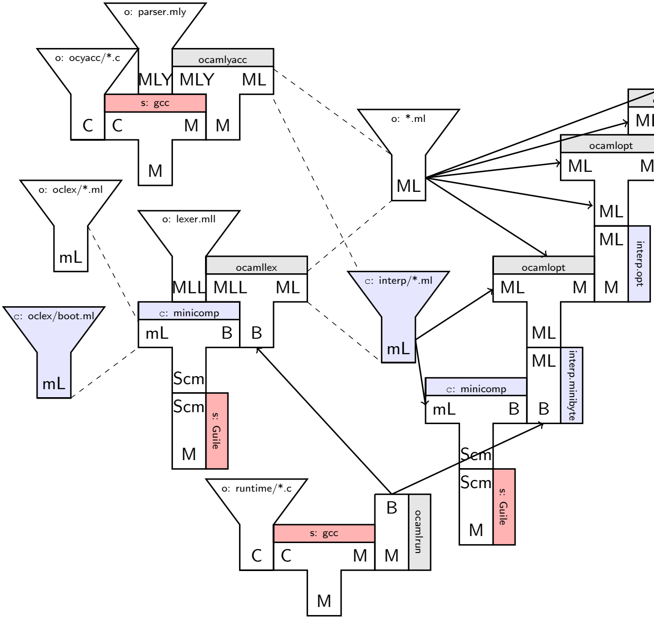
M: machine code.
C: C source code.
ML: OCaml source code.
MLY: OCaml parser definition.
red s: bootstrap seeds

Building OCaml 4.07



B: OCaml bytecode
M: machine code
C: C source code
ML: OCaml source code
MLL: OCaml lexer definition
MLY: OCaml parser definition
o: OCaml compiler sources
red s: bootstrap seeds

Building OCaml with Camlboot



B: OCaml bytecode
M: machine code
C: C source code
ML: OCaml source code
mL: MiniML source code
Scm: Scheme source code
MLL: OCaml lexer definition
MLY: OCaml parser definition
o: OCaml compiler sources
red s: bootstrap seeds
blue c: our Camlboot code

Why interpret `ocamlopt` instead of `ocamlc`?

- A priori: `ocamlopt` is more complicated to interpret: uses objects.
- But `ocamlc` cannot work: it uses `Marshal` to directly write values to the output file.
  $\Longrightarrow$ Incompatible formats: we serialize our representation of the value, not the value. $\Longrightarrow$ `ocamlrun` crashes on the resulting bytecode files.

Scope of our interpreter

- Interprets the *untyped* syntax tree.
- Supports almost all of OCaml.
- A few approximations when the semantics depend on typing.
- Written in MiniML, ≈ 3000 lines of code, uses the parser from the OCaml compiler.

Preview of `interp`: type of values

```
type value = value_ Ptr.t
and value_ =
| Int of int
| Function of case list * env
| String of bytes
| Float of float
| Constructor of string * int * value option
| Record of value ref SMap.t
| Prim of (value -> value)
[...]
```

Preview of `interp`: evaluation function

```
let rec eval_expr prims env expr =
  [...]
  | Pexp_try (e, handlers) ->
   (try eval_expr prims env e
    with InternalException v ->
      (try eval_match prims env handlers (Ok v)
       with Match_fail -> raise (InternalException v)))
```

Why use OCaml (MiniML) instead of Scheme?

- Reuse the OCaml runtime primitives
  $\Longrightarrow$ simplifies the interpreter a lot.
- Writing a parser for the full OCaml language is complex
  $\Longrightarrow$ reuse the existing parser.
- A reference interpreter would be useful to the community.

```
let prims = [
  [...]
  ("caml_md5_chan",
   prim2
     Digest.channel
     unwrap_in_channel
     unwrap_int
     wrap_string);
  [...]
]
```

The MiniML language

- Compiled to OCaml bytecode (ZINC abstract machine): can use runtime primitives, closures are easy to compile.
- No support for: objects/classes, lazy values, first-class modules, type-based disambiguation.
- Functors are generative.
- Deciding whether to support a feature or not:
  - Is it used in the interpreter?
  - Is it less work to support it than to remove its use in the interpreter?

The `minicomp` compiler

- Two-pass compiler, written in Scheme, ≈ 3300 lines of code
- First pass (*lowering*): pattern matching compilation, labeled arguments reordering, records and constructors turned into tagged blocks
- Second pass: compilation to bytecode, direct output to file with backpatching as necessary

Preview of minicomp

```
(define (compile-expr env stacksize expr)
  (match expr
    [...]
    (('LIf e1 e2 e3)
     (let* ((lab1 (newlabel))
            (lab2 (newlabel)))
     (compile-expr env stacksize e1)
     (bytecode-put-u32-le BRANCHIFNOT)
     (bytecode-emit-labref lab1)
     (compile-expr env stacksize e2)
     (bytecode-BRANCH-to lab2)
     (bytecode-emit-label lab1)
     (compile-expr env stacksize e3)
     (bytecode-emit-label lab2)))
    [...]
))
```

DDC for OCaml

- We performed diverse double compilation for OCaml 4.07.1.
- OCaml 4.07.1 is free of trusting trust attacks!

Compilation times

- First: basic build, interpreted `ocamlopt` directly compiles `ocamlc`

	First	Optimized	Parallel
`ocamlrun`	1m		
`interp.minibyte`	2m		
`interp.opt`	not built		
`stdlib.opt`	4h40m		
`ocamlc.opt`	25h40m		
Total	30h23m		

Compilation times

- First: basic build, interpreted `ocamlopt` directly compiles `ocamlc`
- Optimized: compile the interpreter with interpreted `ocamlopt` to speed up further steps

	First	Optimized	Parallel
`ocamlrun`	1m	1m	
`interp.minibyte`	2m	2m	
`interp.opt`	not built	8h56m	
`stdlib.opt`	4h40m	48m	
`ocamlc.opt`	25h40m	4h08m	
Total	30h23m	13h55m	

Compilation times

- First: basic build, interpreted `ocamlopt` directly compiles `ocamlc`
- Optimized: compile the interpreter with interpreted `ocamlopt` to speed up further steps
- Parallel: optimized build, running on 4 cores/8 threads

	First	Optimized	Parallel
`ocamlrun`	1m	1m	1m
`interp.minibyte`	2m	2m	2m
`interp.opt`	not built	8h56m	2h02m
`stdlib.opt`	4h40m	48m	23m
`ocamlc.opt`	25h40m	4h08m	1h31m
Total	30h23m	13h55m	3h59m

Compilation times

- First: basic build, interpreted `ocamlopt` directly compiles `ocamlc`
- Optimized: compile the interpreter with interpreted `ocamlopt` to speed up further steps
- Parallel: optimized build, running on 4 cores/8 threads

	First	Optimized	Parallel
`ocamlrun`	1m	1m	1m
`interp.minibyte`	2m	2m	2m
`interp.opt`	not built	8h56m	2h02m
`stdlib.opt`	4h40m	48m	23m
`ocamlc.opt`	25h40m	4h08m	1h31m
Total	30h23m	13h55m	3h59m

Compilation times are large, but still good enough for reproducibility.

Performance analysis

Lots of inefficiencies that stack upon one another, so slowness is expected.
Compilation times for `interp`:

- With ocamlopt.opt: 1.7s

Performance analysis

Lots of inefficiencies that stack upon one another, so slowness is expected.
Compilation times for `interp`:

- With ocamlopt.opt: 1.7s
- With ocamlopt.byte: 5.8s (3.4x slower)

Performance analysis

Lots of inefficiencies that stack upon one another, so slowness is expected.
Compilation times for `interp`:

- With ocamlopt.opt: 1.7s
- With ocamlopt.byte: 5.8s (3.4x slower)
- With `ocamlopt` interpreted by interp.opt: 2h30mn (1551x slower)

Performance analysis

Lots of inefficiencies that stack upon one another, so slowness is expected.
Compilation times for `interp`:

- With ocamlopt.opt: 1.7s
- With ocamlopt.byte: 5.8s (3.4x slower)
- With `ocamlopt` interpreted by interp.opt: 2h30mn (1551x slower)
- With `ocamlopt` interpreted by interp.minibyte: 13h (5.2x slower)

Performance analysis

Lots of inefficiencies that stack upon one another, so slowness is expected.
Compilation times for `interp`:

- With ocamlopt.opt: 1.7s
- With ocamlopt.byte: 5.8s (3.4x slower)
- With `ocamlopt` interpreted by interp.opt: 2h30mn (1551x slower)
- With `ocamlopt` interpreted by interp.minibyte: 13h (5.2x slower)

The cost of interpretation is far superior than the cost of naïve compilation.

Takeaways from the writing of Camlboot

- Untyped semantics make writing an interpreter *a lot* easier.
  - Untyped semantics are also better for program specifcation and human understanding.
  - Should strive to minimize the parts of OCaml where semantics depend on typing and to minimize their use in OCaml code.
- The operational semantics of parts of the OCaml language are not obvious (module aliases).
- ZINC is really well-designed and makes compiling functionnal languages easy.

Conclusion and future work

- Showed the absence of trusting trust attacks in OCaml 4.07.1.
- Takeaways for the design of OCaml: untyped semantics are good!
- Future work: explore more use cases for an OCaml interpreter (abstract interpretation, differential testing, ...).