Experience Report: debootstrapping the OCaml compiler

ANONYMOUS AUTHOR(S)

It is common for programming languages that their reference implementation is implemented in the language itself. This requires a "bootstrap": a copy of a previous version of the implementation is provided along with the sources, to be able to run the implementation itself.

Those bootstrap files are opaque binaries; they could contain bugs, or even malicious changes that could reproduce themselves when running the source version of the language implementation – this is called the "trusting trust attack". For this reason, a collective project called Bootstrappable was launched in 2016 to remove those bootstraps, by building alternative implementations (old or new) that do not rely on opaque binaries. Those alternative implementations need not be production-ready, it suffices that they are able to run correctly the reference implementation, to obtain a "clean" build that does not depend on the bootstraps; users can then keep using the reference implementation.

We present the work we performed to debootstrap the OCaml compiler, version 4.07. We built a naive interpreter that can interpret the OCaml compiler, and a naive compiler that can compile our interpreter. Using diverse double-compilation, we were able to prove the absence of trusting trust attack in the existing OCaml bootstrap. We believe that the problem of *debootstrapping*, the techniques we used, the issues we faced and those we avoided, could be of interest to the programming-language hackers in our community.

Additional Key Words and Phrases: programming language, compilation, bootstrap, ocaml

1 INTRODUCTION

1.1 Bootstraps and debootstrapping

Programming language designers and implementors devote themselves to creating the best possible programming language, and work hard on turning their language into usable software. What language would they use to implement their ideas? Their own programming language, of course! OCaml is implemented in OCaml, Haskell in Haskell, SML in SML, Scala in Scala, Rust in Rust, the list goes on and on. Some would even go as far as thinking that only *bad* programming languages are not written in themselves. (For a thought-provoking argument *against* this "language implementors should eat their own dogfood" trope, see Laurence Tratt's The Bootstrapped Compiler and the Damage Done.)

But then, beginners ask, how can you run an OCaml program, if you need an OCaml implementation to run the OCaml implementation? How do we solve this chicken-and-egg problem? The answer, we are delighted to explain to them, is to use a *bootstrap*: along the sources of the compiler, we carry a compiled form of the compiler (typically a slightly-older version), which is used to build the compiler itself from sources, and we update this *bootstrap* compiler from time to time. In the old times, a first implementation was written in some lesser programming language, but we replaced it with this bootstrap, which lets us write our compiler code in the best programming language out there.

The problem with this approach is that now we have to carry around this binary blob, version-control it, and we have to *trust* it to correctly compile our programming language. It is a *very opaque* form of mutable state in our development chain. What if the bootstrap compiler has a bug, could this bug somehow reproduce itself within the compiler compiled from the source? What if someone malicious was to intentionally insert malicious logic in the bootstrap compiler, that would compile the source compiler in such a way that it would insert a backdoor in compiled programs, and survive in later bootstraps.

Bugs surviving through bootstraps occur in practice, although rarely. The idea of malicious backdoors hidden in bootstraps is called a "trusting trust attack" [Thompson 1983]. Proof-of-concepts have been implemented, but we do not know if the attack has ever been used in practice.

In response to these worries, an enthusiastic community of free-software developers launched in 2016 the Bootstrappable Builds project, whose goal is to remove all binary bootstraps from language implementations, to get a more trustable operating system that can be built from nothing else than source code[1], first in assembly language, then in increasingly more pleasant programming language. This movement was born out of the earlier Reproducible Builds project, aiming at ensuring fully-deterministic compilations of operating-system packages. Malicious backdoors may still exist, but at least they are all present in the source code.

Reproducible builds have obvious practical benefits: if you distribute binary packages and people don't know if they can trust you, they can compile the software independently and compare the hashes of their binaries with yours, to know for sure that you are distributing the real thing. Debootstrapping builds is taking things much further, and you may consider the worries mostly theoretical in nature; in the present paper, we do not aim to convince you that it is a very important issue, but at least that it is a very *interesting* issue to solve!

(The terminology around *bootstrap* is confusing and confused: what most compiler authors call *a bootstrap* is the build artifact they use to be able to implement their language in itself. What bootstrappable-builds people call *to bootstrap* is to build a toolchain from sources without using (bootstrap) binaries along the way. To avoid any confusion, we will consistently use *a bootstrap* to mean the self-compilation build artifacts, and *to debootstrap* to mean the act of removing any binary bootstrap to allow a build from sources. A *bootstrapped* compiler uses a binary bootstrap, a *non-bootstrapped* or *debootstrapped* compiler does not.)

In this paper, we report on our experience producing `camlboot`, a *debootstrapped* implementation of the OCaml programming language.[2]

Key metric: human work required to debootstrap. If we assume unlimited work resources, debootstrapping is a trivial problem: just port your programming-language implementation to another language that has already been debootstrapped (for example, gasp, C). But this is a massive effort that may never happen in practice – especially as you have to first convince your language implementors to work in a lesser programming language.

Debootstrapping becomes interesting once you consider human effort as a key metric. Can you debootstrap your language implementation in a *reasonable* amount of work?

The idea is to build a *naive* implementation of your programming language. It can be slow, does not necessarily support all language features, its design would not necessarily scale to a full implementation. It can be produced with a much smaller effort and suffices to build your compiler.

Language design also matters. For example, if your programming language satisfies the type-erasure property, then you may implement it without type-checking the programs first, saving the substantial effort of implementing a type-checker.

`camlboot` is the result of several design iterations, but the latest version was implemented from scratch in two human-months of work.

[1] "Source code" is loosely defined as the documents that are designed for human understanding of the program; typically they are the human-written vs. computer-generated artefacts, but the frontier is very blurry. Here "binary" means non-source. Different definitions on what is and is not part of source code result in weaker or stronger conditions for an implementation to be free from non-source bootstrap.

[2] Included in supplementary material.

Diverse double-compilation. If we cut corners to build a *bad* debootstrapped implementation of your language, the maintainers of the better, bootstrapped implementations are not going to throw away their bootstrap and replace it with our stuff. Have we gained anything?

Diverse double-compilation (DDC) [Wheeler 2005] is a technique to use an alternative implementation to gain trust in a reference implementation, proving the absence of trusting trust attack. (It needs reproducible/deterministic compiler builds.) First, we use the bootstrap compiler to build the reference implementation from source, and we check that the resulting binary is identical to the bootstrap binary; let us call this binary the *bootstrapped binary*. Second, we use our debootstrapped implementation to build the reference implementation under test. The result, the *debootstrapped binary*, may be very different from the bootstrapped binary (different or no optimisation, etc., as our debootstrapped compiler produces worse code), but it should have the same semantics. Finally, we use the *debootstrapped binary* to compile the reference implementation again, getting a *final binary*. The final binary was produced without ever using the bootstrap, using the compiler sources from the reference implementation. If it is bit-for-bit identical to the bootstrapped binary, then we have proved the absence of trusting trust attack; if it is not, there may be a malicious backdoor or a self-reproducing bug, but there may also be a reproducibility issue in the toolchain.

We have successfully performed diverse double-compilation using our debootstrapped implementation, and successfully checked that the bootstrap of the reference OCaml implementation, version 4.07 [Leroy, Doligez, Frisch, Garrigue, Rémy, and Vouillon 2018], is free of trusting trust attack.

1.2 Debootstrapping by replaying the bootstraps?

Once you give up on the idea of a full reimplementation in another language, too time-consuming, another idea to build a working compiler from source is to go back in the very far past, to the implementation in another language that was used before bootstrapping was adopted, and then replay all bootstraps. This is not practical:

- Development environments change and your old code does not run on new systems. Working through the right set of virtual machines to use across all points in the implementation history would be a scary amount of tedious work.
- It can be difficult to replay a bootstrap change, due to what we call the problem of the "hard bootstrap". Some implementation changes require to update the bootstrap, and the bootstrap as stored in version-control may *not* be the result of simply compiling the new compiler with the old bootstrap. Consider for example modifying a runtime primitive and its use in the compiler sources at the same time: running the previous bootstrap compiler against the new runtime may not work. What happened is that the compiler developer may have gone through intermediate versions of runtime and compiler changes (in theory one version of each, but in practice several in an implement-test-refine iteration loop), updating the bootstrap compiler in the middle of the process. A some point a satisfying system was reached and committed to version-control, but the sequence itself may not have been versioned. When that happens, it requires expertise in the compiler codebase to "replay" the bootstrap from scratch.
  This is a problem for other reasons (consider someone trying to rebase your changes on their fork of the compiler codebase), and the OCaml compiler is nowadays trying to avoid such hard-to-replay bootstrap changes, but many such changes have been made in the past.

The OCaml compiler version 4.07.1 had 443 bootstrap changes over its development history. Under the extremely optimistic assumption that these two problems combined require 10 minutes of work to replay each bootstrap (the compilation itself typically takes 2-3minutes), one would be looked at 74 hours of tedious work to replay those bootstraps. This is not practical.

Note: The "hard bootstrap" problem is even worse for languages such as Common Lisp or Smalltalk where code is not stored in files but in a live image. For an in-depth discussion of Common Lisp bootstrapping issues, see Rhodes [2008].

1.3 Debootstrapping the OCaml compiler

.

We used the following approach to successfully bootstrap version 4.07.1 of the OCaml compiler:

- Implement an interpreter for OCaml, that we call `interp`. More precisely, it covers the subset of the language used in the reference compiler implementation. `interp` is itself written in a *smaller* subset of OCaml that we intentionally kept small, and call MiniML. This took a few human-weeks.
- Implement a compiler for MiniML in Scheme, that we call `minicomp`. This also took a few human-weeks (for a much smaller set of features than our interpreter supports). The compiler targets the OCaml bytecode, which comes with a pure C virtual machine.
- We can then use our `interp` interpreter to interpret the reference OCaml compiler to build itself, without needing a bootstrap.

One aspect to be careful about is the other build dependencies than the compiler itself, notably the lexer and parser generators. OCaml 4.07 uses a parser generator, `ocamlyacc`, written in C. The lexer generator, `ocamllex`, is written in a small fragment of the language; we extended MiniML to cover it. We still had to "debootstrap" the lexer generator by writing by hand a lexer that could lex `ocamllex`'s own input grammar!

The two-stage process (a naive interpreter compiled by a naive compiler) introduces massive inefficiencies in the build chain: we are spending computer time to save human time. On a specific build, we measured that running `ocamlopt.opt` (the OCaml native compiler, compiled with itself) is 27500x faster than interpreting the same `ocamlopt` compiler using our `interp` interpreter compiled with `minicomp`. In Section 6.1 we discuss these performance gaps in detail. The whole debootstrapping process still runs in human-reasonable time, taking around four hours to complete on a developer machine.

1.4 Results

We were able to debootstrap the OCaml compiler, and check using diverse double-compilation the absence of trusting trust attack in the bootstrapped compiler for OCaml 4.07.1. In total, this effort took less than two human-months. Re-running the full debootstrap chain to compiler the reference compiler takes a few hours. A build recipe for GNU Guix [Courtès 2013] is provided[3], ensuring that anyone in the future will be able to reproduce this result thanks to very precise dependency information on the whole operating system configuration.

As a side-result, we produced a reference interpreter for a large fragment of the OCaml programming language. We ensured that this interpreter is clearly-written; it has explanatory value, and we believe that it will be reusable in many other research venues. For example, having a small and simple reference implementation could be useful for fuzzing-based differential testing of realistic language implementations.

As a take-away for other language communities, we would recommend debootstrapping all implementations (it is challenging, interesting and fun), insist on the practical benefits of preserving type-erasure as much as possible, and encourage language maintainers to write simple reference interpreters of their real-world programming languages.

[3] In supplementary material.

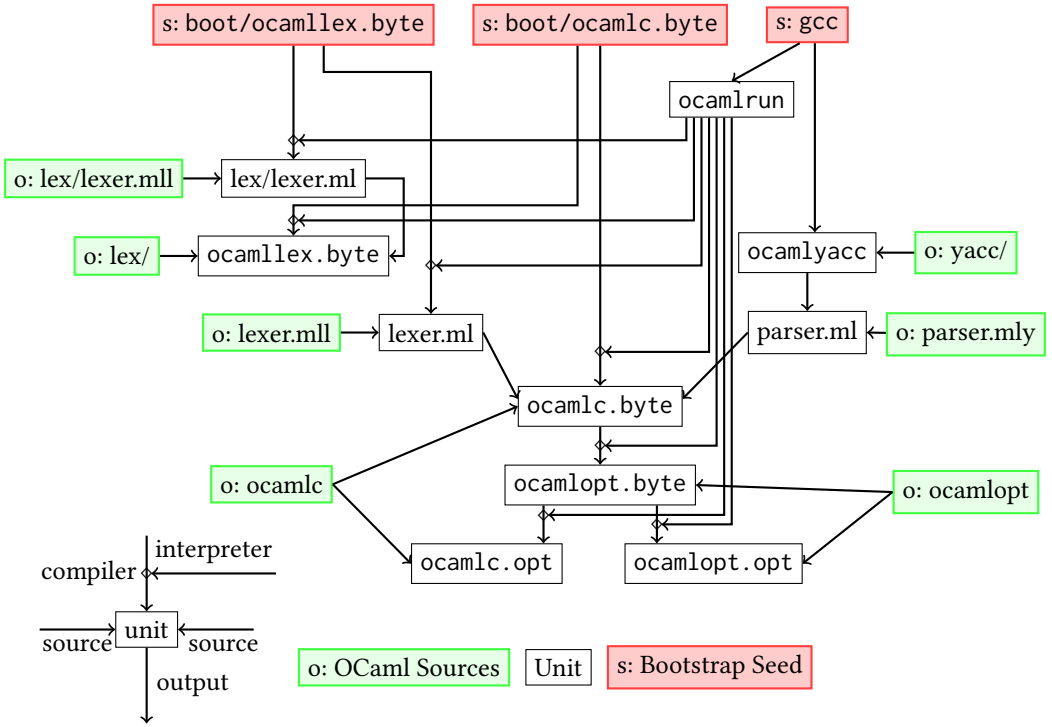

Fig. 1. The usual, bootstrapped build of OCaml 4.07

1.5 Related Work

For reasons of space (Experience Reports have only half the space!), we moved our Related Work section to Appendix A.

2 THE OCAML COMPILER IMPLEMENTATION

OCaml has one reference implementation that evolves along with the language itself, and a few alternative implementations that typically reuse some parts of the reference – to compile to Javascript, etc.

The reference implementation provides *two* compilers: the *bytecode compiler* `ocamlc`, and the *native compiler* `ocamlopt`. `ocamlc` produces executables in a custom portable bytecode format, to be executed by the bytecode interpreter `ocamlrun`. `ocamlopt` produces native binaries directly, which are more efficient (typically 2x-10x) but not portable, larger and longer to compile. Back in the days of Caml Light, the bytecode was famous for its efficiency (among the fastest functional programming languages, due to its clever support for curried application), but nowadays people exclusively use the native compiler in production. Executables produced by the bytecode or native compiler link to a *runtime*, which is a bunch of C code and some assembly that implement low-level features the language relies on, in particular garbage collection.

`ocamlc` and `ocamlopt` are implemented in OCaml. The runtime, as well as `ocamlrun`, are implemented in C code. The OCaml 4.07 compilers also use a parser generator, `ocamlyacc`, implemented in C (but the parsers it generates have an OCaml interface), and a lexer generator, `ocamllex`, implemented in OCaml.

Bootstrap. To bootstrap itself, the OCaml compiler is distributed with bytecode executables for `ocamlc` (the bytecode compiler) and `ocamllex`. They can be executed by `ocamlrun`, which is built from C. To build the compiler from sources they also need to generate an OCaml parser from the yacc grammar, using `ocamlyacc` built from C. Before building the compiler sources it first needs to build the OCaml standard library, distributed with the compiler and used within the compiler sources.

We present in Figure 1 a schematic view of the usual OCaml build plan. We are explicit about the fact that each `foo.byte` binary is interpreted by `ocamlrun`; this expliciteness will be very important to understand the debootstrapping plan in Figure 2 in the next section. This is represented graphically by a "level 2" notation: we draw an arrow from a program P to its output O, to represent program execution, and an arrow between the interpreter of P and this arrow from P to O, to represent interpretation.

In this schema, the cluster involved in building the OCaml lexer generator `ocamllex.byte` is not used to build any further dependencies, as the OCaml language lexer `lexer.mll` is built using the bootstrap binary `boot/ocamllex.byte` instead. Representing the build of `ocamllex.byte` is still important to understand how to refresh the bootstrapped binary when necessary. Note that `lex/lexer.mll` represents the `ocamllex` lexer, used to lex the lexer-description input format (`.mll` file), not to be confused with the OCaml lexer `lexer.mll`.

Note: to simplify the schema we omitted the fact that `ocamlopt.byte`, `ocamlc.opt` and `ocamlopt.opt` all depend on `lexer.ml` and `parser.ml`, same as `ocamlc.byte`.

3 A GLOBAL VIEW OF OUR DEBOOTSTRAPPED BUILD PATH

Our debootstrapped build path starts with both `gcc` and `guile` as seeds. Some parts of the build path of OCaml can be reused: we build `ocamlrun`, `ocamlyacc` and `parser.ml` the same way.

The first change is the construction of the sources of the compiler, more precisely the lexer, which was previously generated by `boot/ocamllex.byte`. We replace it by the version of `ocamllex` packaged with the OCaml sources, and compiled by `minicomp`. There is a difficulty here as well: these sources contain a lexer themselves, so we wrote a lexer by hand `lex/lexer.boot.ml`, that was able to parse `ocamllex`'s own lexer. We use it to generate a lexer for `ocamllex`, and use the result to generate `ocamlc`'s lexer.[4]

Once we have the lexer and the parser, we can compile our interpreter into `interp.minibyte`. We use it to interpret the `ocamlopt` compiler. For performance reasons, the first program we compile with this `ocamlopt` is our interpreter itself (see Section 6).[5] Then, with this natively-compiled interpreter, we interpret `ocamlopt` again to compile the sources of `ocamlc`. With the resulting `ocamlc.opt` compiler, we can finally produce debootstrapped binaries to replace the OCaml bootstrap binaries.

3.1 Extending the build path after OCaml 4.07

At the time this work started, 4.07.1 was the most recent version of OCaml, released in October 2018. The OCaml compiler distribution follows a six-month release cycle, and several versions have been released since, the most recent at the time of writing being 4.12.

[4]The description of the `ocamllex.minibyte` build in the figure is highly simplified. We combine our hand-written lexer with the sources of ocamllex, compile this with `minicomp` into a `lex.boot.byte`. Then we use `lex.boot.minibyte` to compile the reference lexer for `ocamllex`, `lex/lexer.mll`, and combine it with the other sources to produce our final `ocamllex.minibyte`. This two-build process, which is effectively implementing diverse double-compilation for the lexer, was necessary to ensure that the produced lexers are exactly identical to the reference implementation.

[5]In the schema this is represented with two interpretation arrows: `interp.minibyte` interprets the sources of `ocamlc`, but it iself is interpreted by `ocamlrun`.

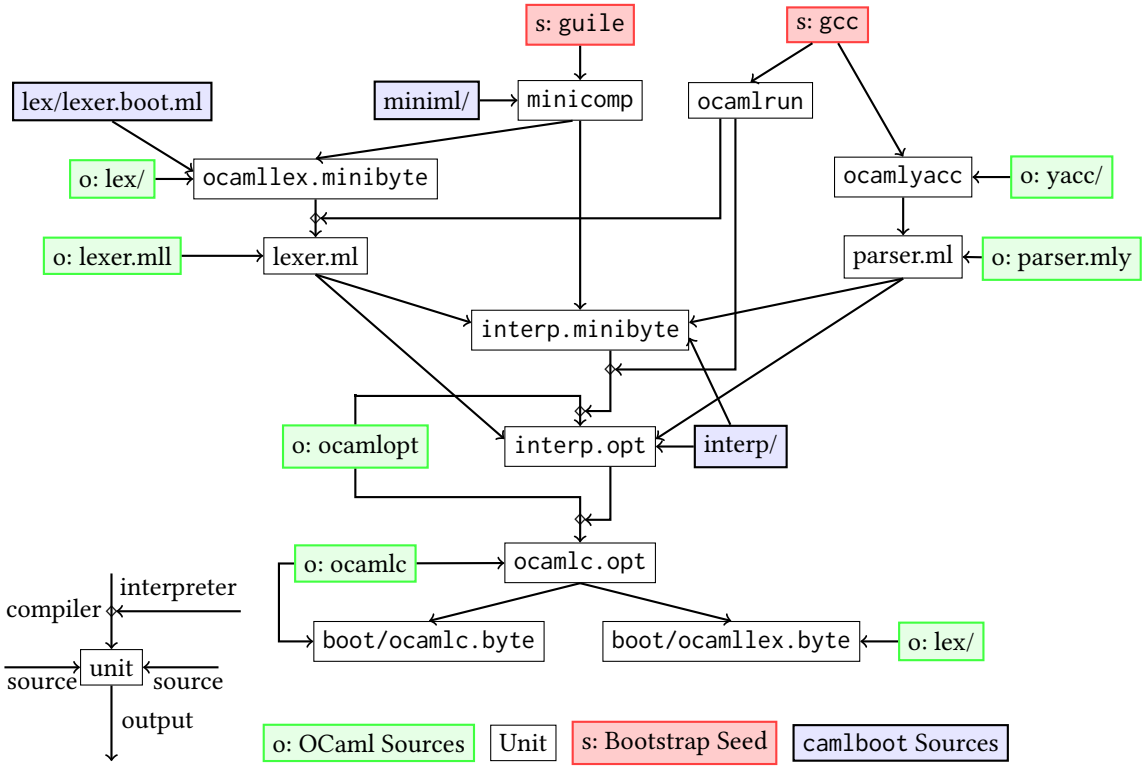

Fig. 2. Our debootstrapped build of OCaml 4.07

We decided to stick with OCaml 4.07 rather than a more recent version, because OCaml 4.08 and later transitioned to Menhir, a parser generator itself written in OCaml, making the debootstrapping path more complex.

We leave a 4.12 build path as future work. There are two possible approaches, one is to try to build 4.12 from our debootstrapped 4.07 implementation, and the other is to update our debootstrapping tools to work with 4.12 directly. The first option corresponds to considering our implementations as "legacy", and we would rather explore the second one. A hybrid approach would be to build Menhir using our debootstrapped OCaml compiler (we checked that it builds with OCaml 4.07, even the recent versions used for the 4.12 parser), but upgrade our interpreter and compiler to support the rest of the 4.12 compiler implementation.

4 INTERP: INTERPRETING OCAML IN MINIML

Our first contribution is `interp`, an interpreter for OCaml written in MiniML, a reasonably-small subset of OCaml. It takes 3000 lines of code, took about four human-weeks of work to write, and it covers all features of OCaml used in the standard library and the compiler, which is almost all the features of the language.[6]

The interpreter is written to be reusable in other projects: it strives to be easy to read and maintain. To get a sense of it, we show the core definition of OCaml "values" in Appendix B.1.

[6]Some runtime libraries not used inside the compiler, such as ephemerons and bigarrays, are not supported by our interpreter.

In Appendix B.2 we give an example of a recent language feature that had been designed with compilation in mind, for which figuring out a source-level operational semantics was not obvious.

Lesson 1. *Maintaining a reference* interpreter *forces you to think about the semantic impact of compiler implementation changes. We recommend it for more informed language evolution.*

4.1 Technical zoom: interpreting `ocamlc` or `ocamlopt`?

The bytecode and native compilers, `ocamlc` and `opt`, share the same frontend (parser, type-checker, pattern-matching compilation, first pass of simplifications/optimizations). They have two different backends, with `ocamlc` having a much simpler backend. To give some numbers, the frontend is about 50K lines of code, the bytecode backend is 4K lines of code, and the native backend about 40K lines of code. Bytecode compilation is also noticeably faster than native compilation, typically twice faster.

When we set out to interpret the OCaml compiler, we thus started interpreting the bytecode compiler : less code to interpret, and the interpreted compiler would be faster to compile. Our attempt was thwarted by an unexpected coupling between the compiler and the language runtime.

In the object files (library archives or executables) produced by the bytecode compiler, there is not only the bytecode instructions for the program, but also various side-data such as tables of constant values and debug information. Most of these file formats are defined (in the compiler implementation) in a precise way, to be parsed by the bytecode interpreter `ocamlrun`: a header with a magic word and section tables in a precise binary format, bytecode instructions in a precise byte encoding, etc. But the constant tables and debug information are serialized using OCaml's built-in, polymorphic pickling/marshaling operation (`input_value` and `output_value`), and inserted as-is in the bytecode object file.

This works fine with the reference implementation of the language, which uses the same implementation of (un)marshalling as `ocamlrun`: they are both using the implementation provided by the OCaml runtime system. But this does *not* work with our interpreter, which interprets the marhsaling primitives differently: it calls the marshalling function on the interpreter representation of the value, which differs from the native representation, and thus we get different rules. If we use our interpreter to interpret the code of `ocamlc`, then the call to `output_value` in `ocamlc` source code (producing the bytecode artifact) will produce a serialized value in a different format, and `ocamlrun` will crash when trying to deserialize this part of the bytecode executable and consume it.

To work around this issue, We could have modified the implementation of `ocamlc` and `ocamlrun` to stop using OCaml's built-in marshaling functions, and instead use a precisely-defined binary serialization format for constant tables and debug information. But this would be a non-trivial change from the reference implementation, and it is unclear that the upstream maintainers would have been willing to integrate it. We would like our debootstrapping process to be maintainable alongside the reference implementation, rather than requiring the application of nontrivial patches to get a debootstrappable compiler.

So we decided to give up interpreting the bytecode compiler `ocamlc`, and interpret the native compiler `ocamlopt` instead. The native compiler produces binaries in its standard system format (ELF on Linux), without any implementation-defined parts. We had to extend our interpreter with a sizeable amount of language features used in the `ocamlopt` codebase but not in `ocamlc`, notably OCaml objects. The backend uses class inheritance to factorize code over several architectures – for example, instruction selection is defined in a main class, that is inherited in each backend to implement architecture-specific refinements.

Lesson 2. *While the implementation of your compiler may of course use various primitives of your language with implementation-defined behavior, it is not a good idea to have its* outputs *artifact*

output use implementation-defined formats. The compiler artifacts format should be defined so that different implementations can easily achieve cross-compatibility.

5 `minicomp`: COMPILING MINIML TO THE OCAML BYTECODE

Our second contribution is `minicomp`, a compiler for MiniML to OCaml bytecode, written in Scheme (more specifically, `guile`). It is comparable in terms of complexity to `interp`, taking about 3300 lines of code and having taken about four human-weeks of work to write. The feature set is more restricted than for `interp`: it does not handle objects, classes, lazy values, first-class modules or format strings; functors are generative and compiled by defunctionalization, and type-based disambiguation of constructor names or record labels is unsupported as well.

For the frontend, `minicomp` uses the `lalr-scm` parser generator, and a handwritten lexer. In the backend, `minicomp` produces OCaml bytecode, so that:

- We can use OCaml primitives, especially the ones used by the generated lexer and parser.
- We get good integration with an efficient garbage collector.
- We get efficient support for closures and currified functions.

An early experiment compiled MiniML to C code linked with the OCaml runtime instead; it produced less efficient code, due to having to register every intermediate value as a GC root and having to check for exceptions at each call. In addition, it did not handle closures, which made programming inside its MiniML subset quite painful.

The compiler itself is divided into two passes: a first pass, *lowering*, simplifies input expressions by compiling pattern matching to lower-level constructs (“switch” and “jump-with-arguments”), handles function application (recognition of tail calls, primitive calls, reordering labeled arguments), and transforms constructors and records into blocks with a fixed integer tag. The second pass compiles these simpler expressions and outputs the result directly to a bytecode file (no intermediate representation of bytecode exists in the compiler), backpatching labels as necessary. Most constructions of the lowered expressions map directly to bytecode, and we only need to take care of the scope of variables (local, belonging to the environment of a closure or global), and to the size of the stack.

Here are a few cases of the second compilation pass:

```
(define (compile-expr env stacksize expr)
  (match expr
    [...]
    (('LTailApply e args)
      (let* ((nargs (length args)))
        (compile-args env stacksize args)
        (bytecode-put-u32-le PUSH)
        (compile-expr env (+ stacksize nargs) e)
        (bytecode-put-u32-le APPTERM)
        (bytecode-put-u32-le nargs)
        (bytecode-put-u32-le (+ stacksize nargs))))
    (('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)))
```

The `compile-expr` function takes as input an environment for locating the variables, the current size of the stack, and a lowered expression to compile. It compiles the input by directly writing bytecode opcodes to the output file, with functions such as `bytecode-put-u32-le`. The functions `bytecode-emit-label` and `bytecode-emit-labref` work by recording the current position for future references to the label, and modifying the output file (using `seek`) to overwrite any previous reference to the label with its actual position.

5.1 Defining the scope of MiniML

We did not define MiniML as a precise subset of OCaml when we started the project, but we refined it iteratively.

First we wrote the interpreter, testing it against the compiler codebase (we compile `interp` with the reference compiler for testing), trying to support all necessary OCaml features with a simple, readable codebase. Then we looked at the features we had used in the `interp` codebase that were *not* supported by `minicomp` – outside the MiniML subset of the time. For each such feature, we had the choice to either add it to MiniML and implement it in `minicomp`, or rewrite the interpreter to stop using this feature, possibly at the cost of more verbose and uglier code.

We believe that it is not a coincidence that our compiler and our interpreter consumed roughly the same amount of effort (about four human-weeks each). When deciding whether to extend the compiler or simplify the interpreter, we considered the effort involved on either end. Repeatedly going for the lowest estimated effort tended to evenly balance the time spent on both parts.

In some imprecise sense, we put a square root of the effort required to build a full implementation, by composing two simpler implementations.

6 COMPILING OCAML WITH OUR INTERPRETER

Once we had all the pieces in place, we could put them together and debootstrap the OCaml compiler. The idea is to *compile* the programs `ocamlc` and `ocamllex` from the reference implementation using the reference `ocamlopt` native compiler, *interpreted* from source by our `interp` interpreter, itself *compiled* into OCaml bytecode by our `minicomp` compiler and then *interpreted* by the reference `ocamlrun` bytecode interpreter, Phew!

In this section, we adopt precise yet concise notations to talk about a particular way to run a particular implementation. For an OCaml program `foo`, we write `foo.opt` for the native binary produced by `ocamlopt`, `foo.byte` for the bytecode binary produced by `ocamlc`, and `foo.minibyte` for the bytecode produced by our naive `minicomp` compiler. "Running" any of those programs means either running the native code directly, or using `ocamlrun` to run the bytecode. We also write f(`foo`) to talk about the action of running `foo` interpreted by an interpreter f. To reformulate the previous paragraph, this section is about running `interp.minibyte`(`ocamlopt`) to compile `ocamlc` into `ocamlc.opt`, and from there produce clean binaries to replace the bootstrapping copies of `ocamlc.byte` and `ocamllex.byte`.

You will not be surprised to hear that naively interpreting a large, complex program is *slow*, and that running that interpreter as a naively-compiled program becomes *really slow*. Therefore, we iterated over several build plans in order to make the experiment fast enough to be reproducible.

	First	Optimized	Parallel
`ocamlrun`	1m		
`interp.minibyte`	2m		
`interp.opt`		8h56m	2h02m
`stdlib.opt`	4h40m	48m	23m
`ocamlc.opt`	25h40m	4h08m	1h31m
Total	30h23m	13h55m	3h59m

Table 1. Timing of our Three Build Plans

All our build plans start by building `ocamlrun` from the C sources of the OCaml compiler distribution, which takes around one minute, and then run `minicomp` to compile `interp` into `interp.minibyte`. This step takes around two minutes – in any case, these times are negligible compared to the time taken for the other steps. The complete timings for the build plans are summarized in Table 1. All times were measured on a machine equipped with an Intel(R) Core(TM) i7-8665U CPU @ 1.90GHz CPU (4 cores, 8 threads) and 16 GB of RAM.

First build plan. Our first complete debootstrapped build took the following steps:

(1) Run `interp.minibyte(ocamlopt)` to compile the OCaml standard library, a dependency of the compiler sources.
(2) Run `interp.minibyte(ocamlopt)` to compile `ocamlc` into `ocamlc.opt`.

At this point we were not completely finished, but we knew we could use `ocamlc.opt` to continue the debootstrap in reasonable time (building the whole compiler codebases from it takes a few minutes at most). However, the total running time was around 30 hours, which we needed to optimize.

Improved build plan. We improved the code generation of `minicomp` slightly, which gave us a 10-20% performance improvement in `interp.minibyte`, and we added an extra step: instead of using `interp.minibyte(ocamlopt)` to compile the standard library then `ocamlc`, we would use it to compile `interp.opt` first, and then use that faster interpreter for the other compilation steps:

(1) Run `interp.minibyte(ocamlopt)` to compile the `interp` into `interp.opt`.
(2) Run `interp.opt(ocamlopt)` to compile the standard library.
(3) Run `interp.opt(ocamlopt)` to compile `ocamlc`.

This extra step shrinks the total build time from around 30 hours to around 14 hours.

Parallelized build plan. None of the many implementations discussed here are doing any active effort to parallelize builds, but compiling a codebase in independent files/modules leads itself naturally to makefile-level parallelism. With parallel builds enabled, we got actually reasonable build times: the total build time shrank from around 14 hours to around 4 hours.

(If the OCaml compiler was implemented in C++, those would be the usual build times! :-)

At this point it is reasonable to expect other people to reproduce the experiment, and check that our scripted diverse double-diverse compilation confirms the absence of trusting trust attack in the reference bootstrap binaries.

6.1 Performance analysis

We piled two bad implementations on top of each other to produce the slowest ever execution of the `ocamlopt` compiler on a human-written medium-sized program. It is interesting to analyse the sources of inefficiencies.

Let us compare the times taken to compile the `interp` codebase, without any parallelism:

- with `ocamlopt.opt`: 1.7s
- with `ocamlopt.byte`: 5.8s (3.4x slower)
- with `interp.opt`(`ocamlopt`): 2h30 (1551x slower than 5.8s with `ocamlopt.byte`)
- with `interp.minibyte`(`ocamlopt`): 13h (5.2x slower than 2h30 with `interp.opt`)

This suggests that the performance gap between interpretation and compilation is much larger than between bad compilation and better compilation.

On other (smaller) compile targets, we observed that

- `interp.byte` is around 2.5x slower than `interp.opt`, and
- `interp.minibyte` is in turn 2.2x slower than `interp.byte`.

It is interesting that the performance cost of bytecode compared to native code is similar to the performance cost of our "naive bytecode generation" compared to the decent optimization and code-generation work of `ocamlc`. This suggests that we could improve performance of our debootstrapping toolchain by improving the bytecode generated by `minicomp`, but the lack of convenient profiling tools for bytecode programs makes it hard to determine what are our main sources of inefficiencies in the produced bytecode.

7 CONCLUSION

Debootstrapping a language implementation by writing new code is a highly rewarding adventure; we encourage our readers to go at it! Our approach was to write an interpreter in the language itself, staying inside a simple enough fragment that we could write a compiler for. Of course, you now also have the possibility of using OCaml to implement your interpreter.

As a side-result, we wrote a simple interpreter for a large subset of OCaml, that already forced us to think hard about the operational semantics of newer language features. We expect that it will find various unplanned use-cases in the future.

We checked the absence of trusting trust attack in the version 4.07 of the OCaml compiler. In the medium-term future, we want to debootstrap more recent versions of OCaml as well, and think about how to maintain our debootstrap toolchain to follow the evolution of the language and its reference implementation.

REFERENCES

Ludovic Courtès. 2013. Functional Package Management with Guix. In *6 th European Lisp Symposium*. 4.

John Hodge. 2016. mrustc: a Rust compiler in C++. https://github.com/thepowersgang/mrustc/

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2018. *The OCaml system release 4.07*.

Jan Nieuwenhuizen. 2016. GNU Mes. https://www.gnu.org/software/mes/

Jeremiah Orians. 2017. stage0. https://github.com/oriansj/stage0

Christophe Rhodes. 2008. SBCL: A Sanely-Bootstrappable Common Lisp. In *Workshop on Self-Sustaining Systems*.

Timothy Sample and Jan Nieuwenhuizen. 2018. Gash. https://savannah.nongnu.org/projects/gash/

Ken Thompson. 1983. Reflections on trusting trust. In *ACM Turing award lectures*.

David Tolnay. 2019. Bootstrapping rustc from source. https://github.com/dtolnay/bootstrap/

David A Wheeler. 2005. Countering trusting trust through diverse double-compiling. In *21st Annual Computer Security Applications Conference (ACSAC'05)*. IEEE, 13–pp.

Andy Wingo, Marius Vollmer, Mikael Djurfeldt, Ludovic Courtés, and Jim Blandy. 1993. GNU Guile. http://www.gnu.org/software/guile/guile.html

Ricardo Wurmus. 2017a. Bootstrapping Haskell: part 1. https://elephly.net/posts/2017-01-09-bootstrapping-haskell-part-1.html

Ricardo Wurmus. 2017b. Building the JDK without Java. https://www.freelists.org/post/bootstrappable/Building-the-JDK-without-Java

A RELATED WORK

The Bootstrappable Projects website describes heroic efforts to debootstrap various parts of the free-software ecosystem. It relies on GNU Guix [Courtès 2013], an operating system distribution that tracks dependencies in a fine-grained way, providing a tree of packages that rely on as few *bootstrap seeds* (non-source forms of programs required to build the system) as possible. Debootstrapping a bootstrapped programming language means that its bootstrap binary needs not be part of the bootstrap seeds anymore.

Guix is itself implemented in GNU Guile [Wingo, Vollmer, Djurfeldt, Courtés, and Blandy 1993], a nice implementation of Scheme. The Guix, Guile and Scheme community at large are strong contributors to the Bootstrappable effort. Scheme is a natural choice for debootstrapping, being arguably the simplest programming language in which you might want to write a programming language implementation.[7]

It is interesting to look at how other languages have approached debootstrapping. A short summary is that very few of the higher-level languages that have been relying on a bootstrap for a long time have been debootstrapped, a few exceptions being C, Rust and Java (but not Scheme!). OCaml is joining a select group with our work.

Debootstrapping a programming language has often been achieved by finding existing alternative implementations in a simpler language. This often means working with legacy software that can only build old versions of the bootstrapped implementation, and then trying to replay the bootstrap chain. In Section 1.2 we explained that this is impractical in OCaml due to the problem of "hard bootstraps", but other communities have been doing a better job avoiding such hard bootstraps, for example by imposing that version N+1 can always be built from version N unchanged. One problematic aspect of this method is the reliance on legacy software at old, unsupported versions – the Bootstrappable community has to maintain those old versions in a working state to keep the bootstrap chain reproducible, for example porting it if we moved to a new computer architecture (RISC-V?) that did not exist at the time it was written.

Rust. The reference Rust compiler, `rustc`, has been debootstrapped [Tolnay 2019], making Rust the most advanced programming language with a good debootstrapping story so far. The debootstrap relies on mrustc [Hodge 2016], a partial reimplementation of Rust in C++. `mrustc` does not build recent versions of `rustc`, but it can build version 1.29.2, released in October 2018. Then 1.29.2 can be used to build 1.30, and so on, until the most recent release 1.50 – `rustc` does not suffer from "hard bootstraps". Building each `rustc` version takes about 3 hours on a good machine, suggesting that the full debootstrapped build would take 2-3 days.

One point of interest is that `mrustc` implements type-checking for Rust programs but not lifetime inference, thanks to the "lifetime-erasure" property that lifetimes do not affect the dynamic semantics of the program. This is the ownership analogue of the type-erasure property we rely on in our own work.

Java. Java has been debootstrapped [Wurmus 2017b] by finding an elaborate path through legacy implementations. Let us quote the beginning of the description of this path in the Bootstrappable page on Java, to give an idea of the work involved:

> In Guix the Java bootstrap begins with Jikes, a Java compiler written in C++. We use it to build a simple version of GNU Classpath, the Java standard library. We chose version 0.93 because it is the last version that can be built with Jikes. With

[7] Our experience confirms that implementing languages in Scheme is sensible. But it also suggests that writing a compiler in a non-statically-type-checked language is a source of constant frustration. Every time you change a detail of an intermediate representation somewhere, random stuff breaks because you did not update its consumers, and tracking them down is work.

> Jikes and this version of GNU Classpath we can build JamVM, a Java Virtual Machine. We build version 1.5.1 because it is the last version of JamVM that works with a version of GNU classpath that does not require ECJ. These three packages make up the bootstrap JDK.
>
> This is sufficient to build an older version of Ant, [...]

Haskell. GHC has not been successfully debootstrapped yet. The most elaborate (failed) attempt so far [Wurmus 2017a] managed to run the `nhc98` using the `hugs` interpreter – implemented in C.

Gcc. `gcc` has been successfully debootstrapped from much simpler implementations. This required two components:

- a C implementation able to build `gcc`, or rather, the latest version of `gcc` implemented in C instead of C++, `gcc` 4.7 (released in 2014), that can then build newer versions of `gcc`. This is provided by GNU Mes [Nieuwenhuizen 2016], a project to provide a Scheme interpreter (in C) and a small C compiler `mescc` (in Scheme) that can host themselves. `mescc` can build TinyCC, which can in turn build the C code of `gcc` 4.7.
- Building `gcc`, or any modern software, also requires various core utilities (`patch`, `sed`), GNU Make, and a working Bash implementation. Building those without a working gcc is hard! The Guix project was able to remove them from the bootstrapping base by relying on Gash [Sample and Nieuwenhuizen 2018], a Guile implementation of Bash and core utilities.

Scheme. To the best of our knowledge, Guile itself is not currently debootstrapped. Many necessary pieces are present in the Guile implementation (it has a Scheme interpreter, written in plain Scheme, that can interpret the Guile compiler, and a plain Scheme interpreter in C that can interpret the interpreter), and the Scheme interpreter of GNU `mes` seems close to be able to run `guile`, but still debootstrapping has not yet been achieved. The issue comes from its macro-expander, which needs a pre-expanded version of itself as a bootstrap seed.

Racket had a macro-expander implemented in C until version 6 included (Racket 6.12 was released in 2018). Version 7 moved to a macro-expander implemented in Racket, which was great for maintainability but introduced a bootstrap.

Down to assembly. There is work ongoing to connect GNU Mes "up" with Guile, and also (mes-m2) to connect it "down" with the stage0 project [Orians 2017], a project to build a tower of x86 assemblers from a single 1KByte binary.

Summary. We found that the approach of writing entirely new implementations for the purpose of debootstrapping is more rarely used – to our knowledge, the only project to have used it before our work is GNU Mes. We hope to inspire more people to do it!

In the general case, creating a "build path" from bootstrap seeds to a language implementation may require a mix of legacy implementations, bootstrap replay, and new alternative implementations. New implementations written for debootstrapping may in turn become legacy applications if they are not maintained actively – and become incompatible with newer versions of the reference implementation. The built paths themselves need to be maintained; they can typically be lengthened by adding extra steps to build more recent versions of the reference implementation (for Rust, for example), or shortened by finding alternative, simpler build paths (by improving `mrustc` or, in our case, `interp` and `minicomp`). There are interesting maintenance challenges in there, and only time will tell how effective the Bootstrappable community is at tackling them.

Debootstrapping build plans are currently evolving rapidly; for example the live-bootstrap projects appears to have recently managed a build of `bash` and `make` without relying on Gash/Guile. The global picture may be much improved by the time you read this articule. But each bootstrapped

programming language, OCaml in our case, still needs to be debootstrapped independently from the rest of the software ecosystem.

B MORE ON INTERP

B.1 A taste of the implementation

Here is the core data definition of OCaml "values".

```
type value = value_ Ptr.t
and value_ =
  | Int of int
  | Int32 of int32
  | Int64 of int64
  | Nativeint of nativeint
  | Fun of arg_label * expression option * pattern * expression * env
  | Function of case list * env
  | String of bytes
  | Float of float
  | Tuple of value list
  | Constructor of string * int * value option
  | Prim of (value -> value)
  | Fexpr of fexpr
  | ModVal of mdl
  | InChannel of in_channel
  | OutChannel of out_channel
  | Record of value ref SMap.t
  | Lz of (unit -> value) ref
  | Array of value array
  | Fun_with_extra_args of value * value list * (arg_label * value) SMap.t
  | Object of object_value
and fexpr = Location.t -> (arg_label * expression) list -> expression option
```

Let us describe some technical details that show up in this type definition. We find them (mildly) interesting in that they show that writing an interpreter reveals interesting aspects of the programming-language semantics.

The `value` defined as `value_ Ptr.t`, where `'a Ptr.t` describes a pointer to a value of type `'a` that may be uninitialized, in which case it can be "backpatched" by assigning it exactly once. This mutable indirection is used to interpret OCaml recursive values.

The two other places where mutability occurs explicitly are `Record of value ref SMap.t`, as record fields may be mutable, and `Lz of (unit -> value) ref`, representing lazy thunks that mutate themselves when forced.

The `Fexpr of fexpr` is used to represent certain compiler primitives, in particular short-circuiting operators `a && b` and `a || b`, whose operational semantics is given by rewriting/desugaring into other expressions, before their arguments are evaluated. In contrast, most primitives are defined by the `Prim of (value -> value)` case that take their arguments as values.

Here are a couple cases of the main evaluation function:

```
let rec eval_expr prims env expr =
  [...]
```

```
| Pexp_ifthenelse (e1, e2, e3) ->
  if is_true (eval_expr prims env e1)
  then eval_expr prims env e2
  else
    (match e3 with
    | None -> unit
    | Some e3 -> eval_expr prims env e3)
| Pexp_try (e, cs) ->
  (try eval_expr prims env e
   with InternalException v ->
     (try eval_match prims env cs (Ok v)
      with Match_fail -> raise (InternalException v)))
```

The `eval_expr` function evaluates expressions to a result; `prims` is an environment to interpreter primitive/intrisics names (it never changes, but its presence breaks a circularity between expression evaluation and primitive evaluation), `env` contains the value of term variables, and `expr` is the expression to evaluate.

In the `Pexp_try` case, evaluating a **`try`** `..` **`with`** `..` expression, we see that a program raising an exception is interpreted by running an exception in the interpreter. We catch this expression, use `eval_match` to match its payload against the exception-handling clauses `cs`; if none of the exception-handling clauses matches the exception, then we propagate the exception to the ambient evaluation context.

Note: `interp` does not embed a parser for the OCaml surface language, it works directly from an parse-tree in the format produced by the compiler parser. This is a natural choice for a reference evaluator, using the compiler as a library to parse in a compatible way, but it also works well in our debootstrapping scenario, where we can execute the compiler parser produced by `ocamlyacc` and feed the resulting parse-trees to our interpreter.

B.2 Technical zoom: module aliases

Writing a reference interpreter will confront yourself to hidden corners of your language design, whose operational semantics you may not be completely aware of. Language changes are proposed by people who are familiar with the reference implementation, a compiler, and may frame changes in terms that are natural from a compilation perspective, without working out the operational semantics precisely.

One such instance for us was “module aliases”. OCaml has an ML module system, which supports nested modules and various forms of module bindings (including modules parametrized over modules). Files passed to the compiler are implicitly mapped into modules, so the file “foo.ml” will have its content put in a module `Foo`. However, these “toplevel module” names must be unique. The linker would reject two modules of the same name to avoid symbol name clashes. So a common idiom is to use long filenames for the toplevel modules, prefixed with the library name, and to provide a short “alias map” module that rebinds them to their shorter name. A library `Lib` may for example be implemented by two files `Lib__Foo.ml` and `Lib__Bar.ml`, with a helper file `Lib.ml` whose content is as follows:

```
module Foo = Lib__Foo
module Bar = Lib__Bar
```

This allows the user to write `Lib.Foo`, unaware of the long module name `Lib__Foo`. (In fact the long module names and the "alias map" module are typically generated by the build system, so they can be considered as implementation details for a poor implementation of namespacing.)

There is still a small glitch with this approach: if `Lib__Bar` actually depends on `Lib__Foo`, can it refer to it through the nice name `Lib.Foo`, or should it use the ugly internal name `Lib__Foo`? Using `Lib` from `Lib__Bar` was originally rejected, as `Lib` itself mentions `Lib__Bar` and OCaml does not allow circular dependencies among files / toplevel modules. But the convenience was too good to pass on, so the language semantics was changed slightly to allow it: the `-no-alias-deps` option, now widely used by default, decides that mentioning a module in the right-hand-side of a module "aliasing" (rebinding), such as `module Bar = Lib__Bar`, does not introduce a strong dependency on this module: it is okay if the module is only present "later" in linking order. Of course, any actual usage of the `Bar` module introduces a strong dependency on its definition `Lib__Bar`, but merely mentioning `Lib` from `Lib__Bar` does not create a circular dependency.

When introduced, this feature was described from a compiler perspective: weak versus strong dependencies, module hash verification, linking order, etc. Giving its actual semantics as a reduction relation (big-step or small-step) turns out to be non-obvious.

Our first attempt was to model aliases as weak values being just variables. Evaluating a module name `M`, for example on the right-hand-side of the binding `module Foo = Lib__Foo`, would just return "the free module name `M`", whether or not `M` had already been provided in the current environment. (We are evaluating modules in linking order, and a module is defined in the environment if it has already been evaluated.) Operations that require accessing the module structure, field access `M.x` for example, would then "force" module names into proper structures, failing if the module is not available in the current module environment. But this does not work in presence of functors, which capture their definition-time module environment. If you pass "the free name `M`" as an argument to a functor, and the functor was defined in an environment where `M` is not available, then the functor will be unable to access the structure of its argument, while the caller of the functor may have had access to the definition of `M`.

In the end we gave up with trying to devise gentle semantics for `-no-alias-deps`. The toplevel modules of an OCaml program are big mutually-recursive definition, where modules can be defined *but not dereferenced* before they have been evaluated. More precisely, toplevel module names evaluate to a mutable cell that starts uninitialized, and gets "backpatched" into a complete structure when the module evaluation succeeds.