Compiler Hacking 101: Runtime Types

A hands-on approach

Nicolás Ojeda Bär, nicolas.ojeda.bar@lexifi.com

FUN OCaml, September 16, 2025, Warsaw

```
$ git clone https://github.com/ocaml/ocaml
$ cd ocaml
$ ./configure --disable-native-compiler \
    --disable-ocamlobjinfo --disable-ocamldoc \
    --disable-ocamldebug --disable-ocamltest \
    --disable-unix-lib --disable-str-lib
$ make -j
$ rlwrap runtime/ocamlrun ./ocaml -I stdlib
```

Compiler Quickstart

The OCaml compiler follows a standard organization in multiple stages, gradually translating the input source code into something that can be executed in the target architecture. Its frontend is composed roughly of the following phases:

- Parsing: source code $\rightarrow$ AST (`Parsetree`, `Parser`)
- Typechecking: AST $\rightarrow$ Type-annotated AST (`Typedtree`, `Typecore`, `Typemod`, etc)
- Translation: Type-annotated AST $\rightarrow$ Untyped Lambda Calculus (`Lambda`, `Translcore`, `Translmod`, etc)

After `Lambda`, there are two possible continuations: generate bytecode for the OCaml VM (`ocamlrun`) or native code, in which case a much longer pipeline comes into play: closure conversion, inlining, instruction selection, register allocation, etc.

Parsing

The AST type is defined in `Parsetree`. Many of you are probably familiar with it because it is used (unfortunately :)) in `ppxlib`. The parser is built using Menhir (see `parser.mly`). The structure of the AST is as follows:

```
type expression =
  { pexp_desc: expression_desc;
    pexp_loc: Location.t;
    ... }
type expression_desc =
  | Pexp_ident of Longident.t loc
  | ... (* one case per type of expression *)
```

Each node is annotated with a location. Paths are represented by `Longident` (show). AST construction helpers in `Ast_helper` (show). Location annotation via `'a loc`, `mkloc`, `mknoloc` (show).

The `-dparsetree` flag can be used to examine the AST. Extension points are represented by `Pexp_extension` (show).

Typechecking

Typechecking is the most complicated part of the compiler (luckily we won't have to mess with it). It takes an AST as before and converts it to a "typed AST" (defined in `Typedtree`) where each term is annotated with its type, detecting any type errors along the way. The structured of the Typed AST is similar to that of the plain AST:

```
type expression =
  { exp_desc: expression_desc;
    exp_type: type_expr;
    exp_env: Env.t;
    ... }
and expression_desc =
  | Texp_ident of Path.t * ...
  | ... (* one case per type of expression *)
```

The field `exp_type` contains the type of the expression, and `exp_env` the environment (an immutable data structure) used to typecheck the expression.

The `-dtypedtree` flag can be used to examine the AST.

Type expressions

Types are defined in `Types` and are represented by `type_expr`. This is a mutable graph-like data structure, where unification is achieved by in-place mutation of shared subterms.

The structure of the type is accessed using `Types.get_desc: type_expr -> type_desc`:

```
type type_desc =
  | Tvar of ... (* 'a *)
  | Tarrow of arg_label * type_expr * type_expr * ... (* l:t1 -> t2 *)
  | Ttuple of type_expr list (* t1 * ... * tN *)
  | Tconstr of Path.t * type_expr list * ... (* (t1, ..., tN) constr *)
  | ...
```

A type constructor has a definition that can be looked up in the current environment using `Env.find_type: Path.t -> Env.t -> Types.type_declaration` (show).

Runtime representation

All OCaml values are represented at runtime by a word-size integer. This integer is, either:

- a value, for integer-like objects (`int`, `char`, `bool`, constant constructors, etc)
- a pointer to a heap-allocated block of a number of fields. Such a block has a byte-sized tag, which specifies the "type" of the block.

See also https://dev.realworldocaml.org/runtime-memory-layout.html.

Manipulate the runtime representation by using the `Obj` module (careful: unsafe!).

- `Obj.repr: 'a -> Obj.t` (into the wilderness), `Obj.obj: Obj.t -> 'a` (back to civilization)
- `Obj.is_int: Obj.t -> bool`, `Obj.is_block: Obj.t -> bool`
- `Obj.size: Obj.t -> int`, `Obj.tag: Obj.t -> int`, `Obj.field: Obj.t -> int -> Obj.t`

(show)

Translation down to `Lambda` language

After typechecking, the typed AST is translated down to a much simpler untyped lambda calculus: pattern matching ($\rightarrow$ decision tree of elementary tests), modules ($\rightarrow$ records of value components), type information is discarded and the uniform value representation is used (a tuple, a record, a constructor are all treated the same way).

The `-dlambda` flag can be used to examine the Lambda representation (show).

Try entering the following in the toplevel with `-dlambda`:

```
let f = function
  | Ok (Ok x) -> x + 1 | Ok (Error _) -> -1 | Error s -> failwith (s ^ s);;
type t = {a: int; b: t option};;
{a = 42; b = Some {a = 101; b = None}};;
[42; 101];;
(42, (101, false));;
module _ = struct type t = int let f = fst type s = char let g = snd end;;
(fst, snd);;
```

(Toy) Runtime Types

Our objective is a "function" `show: 'a -> unit` that will print a textual representation of an arbitrary value on standard output (initially, just integers and strings). We will proceed in 7 steps, corresponding to 4 tasks:

1. Extend the standard library with a definition of type witnesses (step 1)
2. Define an operator `[%t: T]` to build type witnesses (steps 2-5)
3. Write the `show` function (step 6)
4. Insert type witnesses automatically (step 7)

The API we will implement will be unsafe. Building a safe API on top of it is not very difficult, but requires some GADT gymnastics. If time allows, I will explain what one possible such API can look like.

The source code for what follows is available as a series of commits in the `trunk` branch of https://github.com/LexiFi/fun-ocaml-2025, based off commit 53643702ef0345ae5fd1d3f0a10774149f819bf5 of the upstream compiler.

Step 1. Extend the standard library

type.ml

```
type stype =
  | Int
  | String
type 'a ttype = stype
let stype_of_ttype ty = ty
```

type.mli

```
type stype =
  | Int
  | String
type 'a ttype
val stype_of_ttype: 'a ttype -> stype
```

Step 2. The [%t: T] operator

typecore.ml, function type_expect_

```
match sexp.pexp_desc with
| Pexp_extension ({txt="t"}, PTyp ct) -> (* [%t: ct] *)
    let sexp = (* (assert false : ct ttype) *)
      let open Ast_helper in
      let open Location in
      Exp.constraint_
        (Exp.assert_ (Exp.construct (mknoloc (Longident.Lident "false")) None))
        (Typ.constr
          (mknoloc (Longident.Ldot (mknoloc (Longident.Lident "Stdlib__Type"),
                                    mknoloc "ttype"))) [ct])
    in
    type_expect env sexp ty_expected_explained
| Pexp_ident lid ->
```

Step 3. The [%t: T] operator: building type witnesses

translcore.ml

```
exception Unsupported of type_expr
let () =
  Location.register_error_of_exn (function
    | Unsupported ty ->
      Some (Location.errorf "Unsupported type: %a" Printtyp.Doc.type_expr ty)
    | _ -> None
  )

let stype_of_type ty =
  match get_desc ty with
  | Tconstr(path, [], _) when Path.same path Predef.path_int -> Type.Int
  | Tconstr(path, [], _) when Path.same path Predef.path_string -> Type.String
  | _ -> raise (Unsupported ty)
```

Step 4. The [%t: T] operator

Tu "unmask" the trojan horse, we need to be able to decide if a type is of the form `T ttype`.

typeopt.mli

```
val is_ttype: Env.t -> Types.type_expr -> Types.type_expr option
```

typeopt.ml

```
let ttype_path =
  Path.Pdot (Path.Pident (Ident.create_persistent "Stdlib__Type"), "ttype")

let is_ttype env ty =
  match scrape env ty with
  | Some (Tconstr(p, [ty], _)) when Path.same p ttype_path -> Some ty
  | _ -> None
```

Step 5. The [%t: T] operator

translcore.ml

```
let is_ttype_of e =
  match e.exp_desc with
  | Texp_assert _ -> Typeopt.is_ttype e.exp_env e.exp_type
  | _ -> None
let rec const_obj (obj : Obj.t) : structured_constant =
  assert (Obj.is_int obj);
  Const_base (Const_int (Obj.obj obj))
```

translcore.ml

```
and transl_exp0 ~in_new_scope ~scopes e =
  match is_ttype_of e with
  | Some ty ->
    Lconst (const_obj (Obj.repr (stype_of_type ty)))
  | None ->
   match e.exp_desc with
```

```
# Type.stype_of_ttype [%t: int];;
# Type.stype_of_ttype [%t: string];;
# Type.stype_of_ttype [%t: int * string];;
# Type.stype_of_ttype [%t: 'a];;
# let tt : int Type.ttype = [%t: _];;
```

Step 6. The `show` function:

```
let show (t: 'a Type.ttype) (x: 'a) : unit =
  let x = Obj.repr x in
  match Type.stype_of_ttype t with
  | Int -> print_int (Obj.obj x)
  | String -> Printf.printf "%S" (Obj.obj x)
```

The use of `Obj` shows that the current API is unsafe. It is possible to build a safe API on top of this unsafe one using GADTs. Note that the `show` function is nonetheless safe for its callers.

```
# use "show.ml";;
# show [%t: int] 42;;
# show [%t: string] "FUN OCaml";;
```

then ....

```
# show [%t: _] "FUN OCaml";;
```

Wouldn't it be nice to have the compiler insert the `~t:[%t: _]` argument by itself?

Step 7. Automatic insertion of type witnesses

typecore.ml, function collect_apply_args

```
| None ->
    let `Arrow (ty_arg, _, _, _) = arrow_kind in
    match Typeopt.is_ttype env ty_arg with
    | Some _ when not optional && label_name l <> "" ->
    let sarg =
      let open Ast_helper in
      let open Location in
      let open Longident in
      Exp.constraint_
        (Exp.assert_ (Exp.construct (mknoloc (Lident "false")) None))
        (Typ.constr (mknoloc (Ldot (mknoloc (Lident "Stdlib__Type"),
                                    mknoloc "ttype"))) [Typ.any ()])
    in
    sargs, Some (sarg, l), TypeSet.empty, false
    | _ ->
    if TypeSet.mem ty_fun visited then
```

```
# #use "show.ml";;
# show 42;;
# show "FUN OCaml!";;
```

Break

I hope I didn't lose anyone along the way. If you leave this workshop having understood what we did so far, for my part, I will consider it a success.

If you are ambitious and want to go further, there are some TODOs in the next slides that you can try your hand at. I will approach some of them after the break.

TODO

- Write a function `Type.equal: 'a ttype -> 'b ttype -> ('a, 'b) Type.eq option`
- Make use of this feature: `show`, `to_json`, `of_json`, `to_gui`, etc.
- Extend the universe of types covered: labelled tuples, record types, variant types, unboxed types, float arrays and records, inline records, functions, objects, ...
- Handle recursive types: will need to adapt compilation scheme. One possibility is to build a cyclic type witness at compilation time, use `Marshal` to dump it in the generated compilation unit, and unmarshal it at runtime.
- Better error messages (eg include locations)

- Optimize the compilation scheme: share type witnesses, unserialize once per compilation unit if using Marshal.
- If `Type.equal` impelements structural semantics, try implementing nominal semantics instead. You will need to record a unique name for each structural type.
- Implement the following strategy for abstract types: when trying to build a type witness of an abstract type `t` if there is a value of the same name `t` of type `t ttype`, then use it as a witness.

A safe API

The API that we exposed before is unsafe: when writing functions that match on the `stype` datatype, one needs to use `Obj`, which invalidates all soundness guarantees. It is easy to make a mistake and cause a segmentation fault.

One can, however, build a safe API on top of the unsafe one. The proposed interface makes use of some simple GADTs. The implementation is actually pretty simple if you have understood what we have done so far, so I will only give the interface.

Enter the GADTs

```
type _ xtype =
  | Unit : unit xtype
  | Int : int xtype
  | String : string xtype
  | List : 'a ttype -> 'a list xtype
  | Record : 'a Record.t -> 'a xtype
  | Sum : 'a Sum.t -> 'a xtype
  | ...

val xtype_of_ttype: 'a ttype -> 'a xtype
```

Type-safe `show`

```
let rec show: type t. t:t ttype -> t -> unit = fun ~t x ->
  (* Look Ma! No Obj! *)
  match xtype_of_ttype t with
  | Unit -> print_string "()"
  | Int -> print_int x
  | String -> Printf.printf "%S" x
  | List t ->
    print_char '[';
    List.iteri (fun i x -> if i > 0 then print_string ", "; show ~t x) x;
    print_char ']'
  | ...
```

Records

```
module RecordField: sig
  type ('a, 'b) t
  val name: ('a, 'b) t -> string
  val ttype: ('a, 'b) t -> 'b ttype
  val get: ('a, 'b) t -> 'a -> 'b
end

type _ field = Field: ('a, 'b) RecordField.t -> 'a field

module Record: sig
  type 'a t
  val fields: 'a t -> 'a field list
end
```

```
let rec show: type t. t:t ttype -> t -> unit = fun ~t x ->
  match xtype_of_ttype t with
  | Record r ->
    print_char '{';
    List.iteri (fun i (Field rf) ->
      if i > 0 then print_string "; ";
      print_string (RecordField.name rf);
      print_char " = ";
      show (RecordField.ttype rf) (RecordField.get rf x)
    ) (Record.fields r);
    print_char '}'
  | ...
```

Tuples

```
type _ xtype =
  | Tuple: 'a Record.t -> 'a xtype
  | ...
```

```
let rec show: type t: t:t ttype -> t -> unit = fun ~t x ->
  match xtype_of_ttype t with
  | Tuple r ->
    print_char '(';
    List.iteri (fun i (Field rf) ->
      if i > 0 then print_string ", ";
      show ~t:(RecordField.ttype rf) (RecordField.get rf x)
    ) (Record.fields r);
    print_char ')'
  | ...
```

Sums

```
module Constructor: sig
  type ('a, 'b) t
  val name: ('a, 'b) t -> string
  val ttype: ('a, 'b) t -> 'b ttype
  (* constant constructor have 'b = unit, otherwise 'b = tuple type *)
  val project_exn: ('a, 'b) t -> 'a -> 'b
end

type _ constructor = Constructor: ('a, 'b) Constructor.t -> 'a constructor

module Sum: sig
  type 'a t
  val constructors: 'a t -> 'a constructor list
  val constructor: 'a t -> 'a -> 'a constructor
end
```

```
let rec show: type t. t:t ttype -> t -> unit = fun ~t x ->
  match xtype_of_ttype t with
  | Sum sum ->
    let Constructor c = Sum.constructor sum x in
    print_string (Constructor.name c);
    print_char ' ';
    show ~t:(Constructor.ttype c) (Constructor.project_exn c x)
  | ...
```

Thanks! Interested?
https://www.lexifi.com/careers