Runtime Types at LexiFi

25 years reflecting types in OCaml

NICOLÁS OJEDA BÄR, LexiFi, France

One of the most common questions asked by OCaml beginners is "how do I print a value?" LexiFi maintains a fork of the OCaml compiler extended with a form of type reflection, which we call *runtime types*. This feature is used to support generic infrastructure for serialization, type-safe marshalling, dynamic values, user interfaces, database layers, language interoperability, and, yes, printing of more or less arbitrary values.

This report describes the system from the programmer's point of view, with particular emphasis on the representation of types, the user-facing APIs, and their integration with the compiler. We also include several examples of use, compare it with other approaches to generic programming in OCaml, and discuss lessons learned over 25 years of use.

1 INTRODUCTION

Most programs include cases where ordinary OCaml values must be handled by generic infrastructure: printing, serialization, dynamic values, generic user interfaces, database layers, scripting.

The objective of this paper is to describe a system of "runtime types" which has been in use at LexiFi for over two decades and is by now quite stable. The design presented here has proven its utility in practice and has stood the test of time. Most of our interesting internal libraries rely on runtime types in some way. The compiler patch is small, relatively easy to maintain and to port between OCaml versions (we are currently tracking the latest OCaml 5.5 release). It does not change the OCaml compilation strategy in any fundamental way (in particular, it does *not* attach metadata to values).

Many of the choices made along the way were pragmatic and driven by the needs of a particular codebase. An upstream design would probably make different choices. Nonetheless, we believe the design presented here provides useful experience for future work in this area.

The main use of this feature is to define functions whose behaviour is directed by the types of their arguments. Such functions are known as generic, polytypic or type-indexed functions. Here, we will stick to the simpler term "generic functions".

Type introspection in OCaml is a rich subject, and over the years many articles and papers have described different approaches to it (see, e.g., [8]), including earlier versions of our system [1, 3–5]. This paper presents an updated and rather complete description of our approach as it exists today, with an emphasis on its user-facing aspects. Section 2 introduces type witnesses and "implicit" arguments. Section 3 presents the GADT-based view used to define generic functions. Section 4 discusses abstract types and custom behaviour, and Section 5 introduces two companion features: type properties and type paths. Applications are shown in Section 6, and a comparison with some related approaches is given in Section 7. We conclude in Section 8 with the main limitations, lessons from long-term use, and maintenance considerations.

For clarity, some of the interfaces shown in this report are simplified versions of the actual ones: they preserve the essential typing structure while omitting auxiliary arguments, cached information, and details of error handling.

2 RUNTIME REPRESENTATION OF TYPES

The central element is the *type of types*, which in our system is given by an abstract type:

```
type 'a ttype
```

Author's address: Nicolás Ojeda Bär, nicolas.ojeda.bar@lexifi.com, LexiFi, Paris, France.

A value of type `'a ttype` describes the structure of the type `'a`. Not all types can be represented in this way. Supported types include primitive types, products, lists, arrays, options, tuples, sums, records, objects, functions, lazy values, and abstract types. Exotic types, such as GADTs and first-class modules, are not supported.

2.1 Construction of Type Witnesses

We provide combinators for constructing type witnesses manually.

```
val int: int ttype
val list: 'a ttype -> 'a list ttype
val option: 'a ttype -> 'a option ttype
...
```

However, building type witnesses manually is tedious. Moreover, building witnesses for sums and records is particularly cumbersome and typically involves providing "injection" and "projection" functions, which would make the witnesses non-serializable. Also, representing recursive types in a way that is convenient to manipulate is difficult.

With this in mind, we provide a built-in operator for constructing type witnesses automatically. The payload of `[%t: T]` is an arbitrary type expression `T`, and the result is a value of type `T ttype`:

```
let _ : int ttype = [%t: int]
let _ : Gc.stat ttype = [%t: Gc.stat]
let _ : (Gc.stat * int, string) result list ttype = [%t: (Gc.stat * int, string) result list]
```

If a type is not supported, it is rejected at compile time.

```
$ cat ex.ml
type t = Ex : 'a -> t
let _ = [%t: t]
$ ocamlc -c ex.ml
File "ex.ml", line 2, characters 8-15:
2 | let _ = [%t: t]
            ^^^^^^^
Error: The type t cannot be a dynamic type (GADT not supported for dynamic types)
```

The argument `T` of `[%t: T]` must not contain type variables after unification.

```
$ cat var.ml
let _ = [%t: 'a * int]
$ ocamlc -c var.ml
File "var.ml", line 1, characters 8-22:
1 | let _ = [%t: 'a * int]
            ^^^^^^^^^^^^^^
Error: The type 'a * int cannot be a dynamic type (type variable)
```

Importantly, unification can be exploited to avoid having to write explicit type annotations most of the time:

```
val print: 'a ttype -> 'a -> string

let _ = print [%t: _] (Gc.stat ())
```

Here the anonymous type variable "_" is inferred to be `Gc.stat` by unification with the type of the second argument.

Implicit Arguments. We have seen that the use of unification lets us avoid writing explicit type annotations in most uses of the `[%t: T]` operator, essentially writing `[%t: _]` at call sites. An additional mechanism makes this even more ergonomic: implicit arguments.

Concretely, whenever a function has a labelled non-optional argument of type `'a ttype`, the compiler will automatically fill it in at the call site if it is not provided by the programmer.

For example, consider the following signature for our `print` function:

```
val print : t:'a ttype -> 'a -> string
```

With this signature, if the programmer writes

```
print x
```

the compiler will insert the missing argument, as if the programmer had written

```
print ~t:[%t: _] x
```

allowing, in turn, the missing argument to be inferred by unification.

Type witnesses can still be passed explicitly. This is useful when defining a polymorphic wrapper, which receives a witness and propagates it downwards:

```
val print : t:'a ttype -> 'a -> string
val log   : t:'a ttype -> 'a -> unit

let log ~t x = prerr_endline (print ~t x)
```

At the call site, the programmer can still simply write `log x`, and the compiler will synthesize the missing argument. Inside its polymorphic definition, however, the type `x` is not known, so a witness cannot be synthesized; `log` must propagate its explicit `~t` argument to `print`.

The choice of using labelled arguments for this mechanism was motivated by the desire to avoid interfering with partial application of ordinary arguments (it also fits naturally with the implementation of labelled arguments in the compiler).

A similar mechanism has been proposed for implicit source positions in standard OCaml [7]. In that proposal, a distinguished optional argument is automatically filled with the location of the call site.

The construction and treatment of witnesses for abstract types is discussed in Section 4.

2.2 Type Equality Testing

Type witnesses can be used to test equality of the underlying types.

```
type (_, _) eq = Eq : ('a, 'a) eq

val ttypes_equality : 'a ttype -> 'b ttype -> ('a, 'b) eq option
```

For example, this is how one could test whether a value is an `int`, at runtime:

```
let as_int (type a) (t : a ttype) (x : a) : int option =
  match ttypes_equality t [%t: int] with
  | Some Eq -> Some x
  | None -> None
```

The `Eq` branch refines the type checker's knowledge from "`a` is some type" to "`a = int`". This is the basic pattern repeated throughout the user-facing APIs: dynamic tests return typed evidence, and the evidence is represented by GADTs.

3 PATTERN MATCHING ON TYPES

While testing for equality against a given type is useful, it is not enough to inspect the structure of a type. For example, we would like to be able to answer the question: "is this type a list, and if so, what is the type of its elements?"

To answer such questions, the system provides a GADT-based view of types, allowing one to pattern match on the shape of a type. This view is represented by the type constructor `'a xtype`.

In order to inspect an `'a ttype` we must therefore convert it to an `'a xtype`:

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

The type `'a xtype` is a GADT[1], where each constructor corresponds to a specific type, and the type parameter `'a` is refined accordingly.

```
type 'a xtype =
  | Unit     : unit xtype
  | Bool     : bool xtype
  | Char     : char xtype
  | Int      : int xtype
  | Float    : float xtype
  | String   : string xtype
  | Option   : 'b ttype -> 'b option xtype
  | List     : 'b ttype -> 'b list xtype
  | Array    : 'b ttype -> 'b array xtype
  | Lazy     : 'b ttype -> 'b Lazy.t xtype
  | Function : string * 'b ttype * 'c ttype -> ('b -> 'c) xtype
  | Sum      : 'a Sum.t -> 'a xtype
  | Tuple    : 'a Record.t -> 'a xtype
  | Record   : 'a Record.t -> 'a xtype
  | Abstract : string * stype list -> 'a xtype
  | Prop     : (string * string) list * 'a ttype -> 'a xtype
  | Object   : 'a Object.t -> 'a xtype
```

The `'a xtype` representation is the main tool used to write type-directed functions. If pattern matching reveals that we are in, for example, the case `List t`, then the value being described has type `'b list` for some `'b`, and the API also provides a type witness `t` for the element type `'b`.

The `Abstract` constructor is used to represent nominal abstract types and will be discussed in Section 4.2. The `Prop` constructor is used to carry type properties and will be discussed in Section 5.1.

A small generic printer illustrates the idiom:

```
let rec print : type a. t:a ttype -> a -> string = fun ~t x ->
  match xtype_of_ttype t with
  | String   -> x
  | Int      -> string_of_int x
  | List t   -> "[" ^ String.concat "; " (List.map (print ~t) x) ^ "]"
  | Tuple r  -> print_tuple r x
  | Record r -> print_record r x
  | Sum s    -> print_sum s x
  | ...
```

The important point is that `x` remains statically typed in every branch. No unsafe cast is needed in the `String`, `Int`, or `List` cases; the GADT constructor has already justified the refinement.

[1] Incidentally, this use case was one of the motivations for the introduction of GADTs in OCaml, see [2].

3.1 Tuples and Records

Records carry more structure than a single `xtype` constructor can comfortably expose. The system therefore provides descriptor modules: `Record`, `RecordField`, `Sum`, and `Constructor`. Their APIs are existential where necessary and typed where possible.

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

type 's has_record_field = Field : ('s, 't) RecordField.t -> 's has_record_field
type 's field_builder = { mk: 't. ('s, 't) RecordField.t -> 't }

module Record : sig
  type 's t
  val fields : 's t -> 's has_record_field list
  val build  : 's t -> 's field_builder -> 's
end
```

The existential `Field` wrapper lets a heterogeneous list contain all fields of the same record, even though each field has a different result type. After unpacking a field, the accessor is fully typed:

```
let print_record (type s) (r : s Record.t) (x : s) =
  let print_field (Field f) =
    RecordField.name f ^ "=" ^ print ~t:(RecordField.ttype f) (RecordField.get f x)
  in
  "{" ^ String.concat ";" (List.map print_field (Record.fields r)) ^ "}"
```

Tuples reuse the same `'s Record.t` description as records, with each tuple component represented as a field. For an ordinary tuple the field names are empty; for a labelled tuple they are the corresponding component labels.

The function `Record.build` performs the inverse operation. It calls a polymorphic callback once for each field and assembles the returned values into a record. Because the callback is polymorphic, it must be able to produce a value of the type described by any field descriptor. This is useful, for example, when defining generic default values or decoding a record from an untyped representation.

3.2 Sums

Sum types follow the same discipline:

```
type 's has_constructor =
  Constructor : ('s, 't) Constructor.t -> 's has_constructor

module Constructor : sig
  type ('s, 't) t
  val name        : _ t -> string
  val ttype       : (_, 't) t -> 't ttype
  val project     : ('s, 't) t -> 's -> 't option
  val project_exn : ('s, 't) t -> 's -> 't
  val inject      : ('s, 't) t -> 't -> 's
end
```

A constant constructor is considered to have an argument of type `unit`; a constructor with several arguments is considered to have a single argument of tuple type; and a constructor with an inline

record argument is represented by a record descriptor. This uniform view lets client code inspect and construct variants while preserving the type of the constructor payload.

```
let print_sum (type s) (s : s Sum.t) (x : s) =
  let Constructor c = Sum.constructor s x in
  Constructor.name c ^ " " ^ print ~t:(Constructor.ttype c) (Constructor.project_exn c x)
```

3.3 Objects

Object types follow a simpler version of the same descriptor pattern. Their runtime description enumerates the object's methods, whose result types may be different and are therefore hidden behind an existential wrapper:

```
module Method : sig
  type ('s, 't) t
  val name  : _ t -> string
  val ttype : (_, 't) t -> 't ttype
  val call  : ('s, 't) t -> 's -> 't
end

type 's has_method =
  Method : ('s, 't) Method.t -> 's has_method

module Object : sig
  type 's t
  val methods : 's t -> 's has_method list
end
```

After unpacking a `Method`, client code can obtain its name and result type, or call it on an object of the corresponding type. As with record fields, the result remains statically typed. Method descriptors support inspection and invocation, but not construction or update of objects.

4 ABSTRACT TYPES

The previous section described how we can inspect the structure of a type. This works well for types whose definition is public. For abstract types, however, the structure is deliberately hidden, and one cannot in general inspect its shape. In this section we explain how the system handles abstract types and the different techniques that can be used to attach custom behaviour to them.

4.1 Transparent Abstract Types

Sometimes a module wants to keep a type abstract, but still provide a "concrete" witness for it, to allow transparent generic programming. There is an important naming convention to support this case.

If the compiler needs a witness for an abstract type `Mod.t`, and a value named `Mod.t` of type `Mod.t ttype` exists, then that value is used as the witness. This lets modules expose the runtime representation of their own abstract types while keeping the type abstract. Note that semantically this is equivalent to exposing the representation of the type, but nominally the type stays abstract.

```
module Mod : sig
  type t
  val t : t ttype
end = struct
  type t = A of int | B
  let t = [%t: t]
end
```

```
8
9  let x = [%t: Mod.t -> Mod.t]
```

This can also be used for type variables, by making use of locally abstract types:

```
1  let result : type t s. t ttype -> s ttype -> (t, s) result ttype =
2    fun t s -> [%t: (t, s) result]
```

This feature can be used to program combinators that build type witnesses at runtime. A witness for an unconstrained type variable cannot be synthesized by the compiler; it must first be supplied explicitly, as in the example above.

4.2 Nominal Abstract Types

Abstract types which do not export their representation are represented nominally:

```
1  type stype = Ttype : 'a ttype -> stype [@@unboxed]
2
3  type _ xtype =
4    | ...
5    | Abstract : string * stype list -> 'a xtype
```

Here the `string` payload is the "name" of the abstract type, and the existentially quantified values in the `stype list` represent its type parameters (if any). Thus, only abstract types to which a global name can be assigned can be represented. In particular, abstract types defined in functor arguments or first-class modules are not accepted:

```
1  $ cat func.ml
2  module F (X : sig type t end) = struct
3    let _ = [%t: X.t]
4  end
5
6  $ ocamlc -c func.ml
7  File "func.ml", line 2, characters 10-19:
8  2 |   let _ = [%t: X.t]
9                ^^^^^^^^^
10 Error (alert not_a_global_abstract_type): *FUNARG*.X.t
```

An ordinary type alias has no separate nominal identity: for example, `type t = int list` is represented as `int list`. A distinct identity arises when a type is exposed abstractly under a global name.

The identity of the abstract type (the pair consisting of the name and its parameters) uniquely identifies the instance of the abstract type and is marshallable, so it can be used as a key in efficient hash tables for registering type-specific behaviour.

```
1  val hash_ttype: 'a ttype -> int
2
3  module TypeTbl = Hashtbl.Make (struct
4    type t = string * stype list
5
6    let equal (n1, ps1) (n2, ps2) =
7      String.equal n1 n2 &&
8      List.length ps1 = List.length ps2 &&
9      List.for_all2 (fun (Ttype t1) (Ttype t2) -> Option.is_some (ttypes_equality t1 t2)) ps1 ps2
10
11   let hash (n, ps) =
12     Hashtbl.hash (n, List.map (fun (Ttype t) -> hash_ttype t) ps)
13 end)
```

4.3 Registering Custom Behaviour

There is no central registry built into the runtime-type system. Each generic library can implement its own registration mechanism in ordinary OCaml, as needed, for instance using hash tables keyed by the identities of abstract types.

With tables such as `TypeTbl` in hand, one can implement custom behaviour for abstract types in generic functions. For example, a generic printer could be extended to support printing of abstract types as follows:

```
type printer = Printer : 'a ttype * ('a -> string) -> printer

let custom_printers = TypeTbl.create 16

let register_printer (t : 'a ttype) (printer : 'a -> string) =
  match xtype_of_ttype t with
  | Abstract (name, params) ->
      TypeTbl.add custom_printers (name, params) (Printer (t, printer))
  | _ ->
      invalid_arg "not an abstract type"

let print (type a) ~(t : a ttype) (x : a) : string =
  match xtype_of_ttype t with
  | Abstract (name, params) ->
      begin match TypeTbl.find_opt custom_printers (name, params) with
      | Some (Printer (t', printer)) ->
          begin match ttypes_equality t t' with
          | Some Eq -> printer x
          | None -> assert false
          end
      | None -> "<abstract>"
      end
  | ...
```

Registration makes it possible to preserve abstraction without reducing all generic operations to an uninformative default. The module defining an abstract type can install suitable behaviour without revealing its representation, and the same registration is then found whenever an independently constructed witness for that type is encountered.

4.4 Parametric Abstract Types

Registration is not limited to a single closed type. A generic function may also register behaviour once for every instance of a parametric abstract type constructor such as `'a M.t`. The library first needs to recognize an application of the constructor and recover the witness for its parameter. In simplified form, the corresponding matcher has the following interface:

```
module type ABSTRACT_1 = sig
  type 'a t
  val t : unit t ttype
end

module Abstract_1_matcher (T : ABSTRACT_1) : sig
  type _ is_t = Is : 'a ttype -> 'a T.t is_t
  val is_t : 'b ttype -> 'b is_t option
end
```

The witness for `unit t` serves to identify the nominal type constructor; the particular argument `unit` is otherwise immaterial.

If `is_t t` returns `Some (Is parameter)`, the GADT match establishes that the type represented by `t` is an application of `T.t`, while `parameter` describes its argument. A generic printer can therefore register behaviour for the entire constructor rather than separately for `int M.t`, `string M.t`, and every other instance. For example, a simplified registration interface is:

```
module type CUSTOM_PRINTER_1 = sig
  type 'a t
  val t : unit t ttype
  val print : ('a -> string) -> 'a t -> string
end

val register_printer_1 : (module CUSTOM_PRINTER_1) -> unit
```

The printer supplied for the element type is obtained recursively by the generic function. The same scheme extends to constructors with more parameters and is used by other generic libraries, not only by printing.

5 COMPANION FEATURES

In this section we describe two features that complement the runtime representation of types: type properties and type paths. These features are not strictly necessary for the basic functionality of runtime types, but they are useful in practice (especially the first one) and are used extensively in our codebase.

5.1 Type Properties

Often type-directed functions need their behaviour to depend on more than the shape of a type. For example, a GUI renderer might want to know that a field is read-only, or that a constructor should be displayed with a particular label. A validator might want to know that a field is required, or that a number should be rounded to a certain number of decimal places. A serializer might want to know that a field should be omitted if it is empty, or that a string should be escaped in a certain way.

To support these and other use cases, it is possible to annotate type definitions with *type properties*. These are string-valued key-value pairs that "flow" along with the type and can be retrieved using the same `xtype` API.

Within a `[@t ...]` or `[@@t ...]` annotation, several properties can be separated by semicolons, as in `[@t key1 = "value1"; key2 = "value2"]`. A property written without an explicit value, as in `[@t key]`, is shorthand for a property whose value is the empty string, and is conventionally used as a flag.

These annotations extend the type algebra; they are not merely metadata stored beside an otherwise unchanged type declaration. Properties can be attached to type expressions, fields, constructors, and record or variant declarations:

```
type amount = float [@t decimals = "2"]
type raw_amount = float

type trade = {
  notional : amount [@t group = "Economics"];
  currency : string [@t autocomplete = "currency"];
} [@@t label = "Trade"]

type side =
  | Buy  [@t label = "Buy-side"]
  | Sell [@t label = "Sell-side"]
```

The first two declarations define distinct types as far as runtime-type inspection is concerned. They are, however, compatible for unification, so an `amount` can be used as a `float` without an explicit conversion. Properties thus add information which generic functions can observe without introducing new incompatibilities into ordinary OCaml code.

At runtime, properties appear either as a `Prop` wrapper in `xtype` or as metadata on fields and constructors, via `RecordField.props` and `Constructor.props`.

When testing type witnesses for equality using `ttypes_equality`, properties are taken into account. It is also possible to test for equality "modulo type properties", using

```
val ttypes_equality_modulo_props : 'a ttype -> 'b ttype -> ('a, 'b) eq option
```

The `xtype` view exposes type properties using the `Prop` constructor:

```
type _ xtype =
  | ...
  | Prop : (string * string) list * 'a ttype -> 'a xtype
```

Generic functions defined by pattern matching on `_ xtype` values can thus accumulate properties as they traverse the type structure and interpret them according to their needs.

5.2 Type Paths

Type paths can be used to refer to subparts of a value in a type-safe manner. They are closely related to the notion of a "lens" in other functional programming languages. Paths from a value of type `'a` leading to a value of type `'b` are represented by the type:

```
type ('a, 'b) type_path
```

Here `'a` is the root type and `'b` is the type reached by the path. A dedicated operator `[%p P]` builds such values from path-like expressions `P`:

- The empty expression, of type `('a, 'a) type_path`.
- `f` selects a record field `f`.
- `(C)` selects the payload of a constructor `C`.
- `(i/n)` selects the `i`-th element of a `n`-tuple.
- `[i]` selects the `i`-th element of a list.
- `[|i|]` selects the `i`-th element of an array.
- `P1.P2` composes two paths, first following `P1` and then `P2`.

So for example,

```
[%p M.foo.(Bar).(0/2).[0]]
```

defines a type path that starts at a record type defined in `M` that contains a field named `foo`, whose type is a sum type containing a constructor `Bar` with payload a pair whose first element is a list, and the path selects the first element of that list.

Besides being constructed directly by the user, type paths can be obtained programmatically from the `'a xtype` representation:

```
module RecordField : sig
  ...
  val path : ('s, 't) t -> ('s, 't) type_path
end

module Constructor : sig
  ...
  val path : ('s, 't) t -> ('s, 't) type_path
end
```

Type paths can be used for typed extraction and update:

```
val extract : t:'a ttype -> ('a, 'b) type_path -> 'a -> ('b ttype * 'b) option
val patch   : t:'a ttype -> ('a, 'b) type_path -> 'a -> ('b -> 'b) -> 'a option
```

Type paths are particularly useful in generic UI code, as they provide a way to refer to subparts of the UI (which correspond to subparts of the underlying data structure) in a type-safe manner. This allows generic operations such as “disable a part of a form” or “apply a patch to a piece of UI”.

6 APPLICATIONS

Here we briefly describe some of our main applications of the preceding machinery.

6.1 Generic Printing

Generic printing is one of the simplest applications of runtime types:

```
val show: t:'a ttype -> 'a -> string
```

The implementation follows the structure exposed by `xtype`, printing record fields, tuple components, and sum constructors recursively. The small printer in Section 3 illustrates the basic pattern. Type properties can control details of the output, while printers registered for particular types handle abstract or domain-specific values. In practice, this gives us a convenient default representation for logging and diagnostics without requiring a printer to be derived or maintained for every application type.

6.2 Dynamic Values

A runtime type witness can be packaged together with a value to form a dynamic value:

```
type dyn = Dyn : 'a ttype * 'a -> dyn
```

Dynamic values can be stored in heterogeneous data structures while retaining enough information to inspect them generically or recover a value at its original type. Recovery uses runtime-type equality and requires no unsafe cast:

```
let of_dyn : type a. t:a ttype -> dyn -> a option = fun ~t (Dyn (t', x)) ->
  match ttypes_equality t t' with
  | Some Eq -> Some x
  | None -> None
```

Compiler-generated witnesses and implicit arguments make both packaging and recovery lightweight at call sites. Dynamic values are useful at boundaries where the set of types is not known statically, while preserving type safety when a value is retrieved.

6.3 Variants

Variants are an untyped representation of OCaml values. They provide a common intermediate language for serialization, language interoperability, and other generic operations. In practice, variants have become a useful boundary between typed application code and external components. Textual persistence, migration tools, generic search, and other generic data-processing tools can all operate on the same representation.

```
type variant =
  | V_unit
  | V_bool of bool
  | V_int of int
  | V_float of float
  | V_string of string
```

```
7     | V_tuple of variant list
8     | V_list of variant list
9     | V_array of variant array
10    | V_option of variant option
11    | V_record of vrecord
12    | V_constructor of string * variant option
13    | V_variant of variant
14    | V_lazy of variant Lazy.t
```

The two main operations convert between typed OCaml values and this common representation:

```
val variant: t:'a ttype -> 'a -> variant
val of_variant: t:'a ttype -> variant -> 'a option
```

The forward conversion is a generic traversal of the runtime type; the reverse conversion uses the same witness to check the shape of the untyped value while reconstructing a typed OCaml value.

Abstract types require custom conversion code. This is a recurring pattern in our libraries: the generic operation handles the structural cases, while a type-indexed hook handles domain-specific abstractions.

```
val register_abstract:
  t:'a ttype ->
  to_variant:('a -> variant) ->
  of_variant:(variant -> 'a option) ->
  unit
```

Type properties are used to guide the work of `of_variant` and make it robust against common changes to persisted data:

- `[@t of_variant_default = "..."]` requests a default value, in variant syntax, when a field is absent from the input variant.
- `[@t of_variant_old_name = "..."]` lets a renamed field or constructor still be read under its former name.
- `[@t of_variant_remove_old_fields]` allows fields which have been removed from the current record to be ignored when reading old data.

Together these properties provide a lightweight scheme for backwards compatibility as data structures evolve, without requiring a separate migration for every small change.

6.4 JSON

```
val to_json: t:'a ttype -> 'a -> Mlfi_json.t
val of_json: t:'a ttype -> Mlfi_json.t -> 'a option
```

The default JSON mapping is structural: records become objects, lists and arrays become arrays, and variants use a conventional object encoding for the constructor name and its arguments. Type properties can alter local details of the mapping, for example by encoding constant constructors as strings instead of objects. A type-specific pair of conversions can also be registered for a closed abstract type, record type, or sum type; parametric abstract type constructors have a corresponding registration interface.

Interpreting a runtime type need not add the cost of a complete type traversal to every encoded value. Partial application to the runtime-type argument builds a closure specialized to that type, and the derived code for nested descriptors is cached. This is an instance of lightweight staging in generic programming [9]: the first stage interprets the type; the resulting closure processes values.

Custom conversions can be installed globally or supplied locally in a conversion context. The latter is useful when a type has different external encodings in different protocols; the former is convenient when one encoding is canonical throughout the application.

```
val register_conversion:
  t:'a ttype ->
  to_json:('a -> Mlfi_json.t) ->
  of_json:(Mlfi_json.t -> 'a option) ->
  unit
```

6.5 Graph Visualization

Runtime types are also useful for diagnostics. The following function produces a Graphviz DOT representation of any supported value:

```
type decision =
  | Default
  | Prune
  | Replace : dyn -> decision

type node     = { label: string; children: dyn list }
type override = { override: 'a. 'a ttype -> 'a -> decision }

val to_dot: ?override:override -> t:'a ttype -> 'a -> string
```

The traversal follows records, sums, tuples, arrays, and lists. Physical sharing and cycles are detected and represented explicitly. Primitive types, functions, objects, and nominal abstract values are represented as leaves.

The optional `?override` parameter is a polymorphic callback which can prune a subtree or replace the value and its runtime type. The `dyn` type is that of Section 6.2. If the underlying type of the `dyn` is `node`, then it is interpreted to specify a subgraph with a custom label and children. This gives callers a way to omit or abbreviate certain parts of the value being displayed to avoid cluttering the resulting graph.

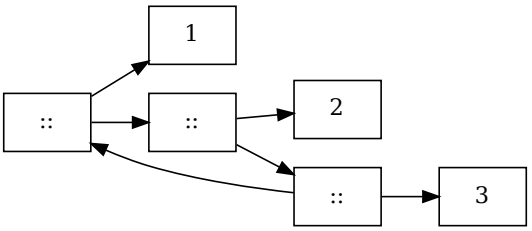


Fig. 1. The cyclic list `let rec xs = 1 :: 2 :: 3 :: xs`.

6.6 QuickCheck

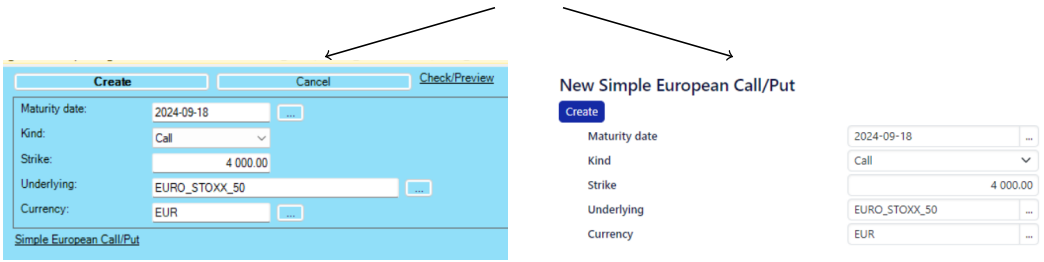
Runtime types can also be interpreted as generators for property-based testing. The generic generator recursively handles primitive types, options, lists, arrays, records, and sums; applications may supply additional generators for abstract or domain-specific types.

```
1  type 'a gen = int -> Random.State.t -> 'a
2  type 'a shrinker = 'a -> 'a list
3
4  type ugen = Gen : 'a ttype * (int -> 'a gen) -> ugen
```

```
type few_digits = float [@t few_digits]

type call_put   = Call | Put

type parameters = {
  maturity_date : Mlfi_date.t;
  kind          : call_put;
  strike        : few_digits;
  underlying    : Equity_underlying.equity_underlying;
  currency      : Mlfi_currency.t;
}
```



Native Web

Fig. 2. Two user interfaces derived from the same record type.

```
val generate: t:'a ttype -> ugen list -> 'a gen
val shrink: t:'a ttype -> 'a shrinker

val run: int -> 'a gen -> ?shrink:'a shrinker -> ('a -> bool) -> (unit, 'a) result
```

This is useful for obtaining generators for ordinary product and sum types with no per-type boilerplate. Fully automatic generation is sometimes *too* random for some use cases. Custom generators are therefore passed as a heterogeneous list, and runtime-type equality recovers the type of a matching generator safely:

```
let rec find_custom: type a. ugen list -> a ttype -> (int -> a gen) option = fun l t ->
  match l with
  | [] -> None
  | Gen (u, fu) :: tl ->
      begin match ttypes_equality t u with
      | Some Eq -> Some fu
      | None -> find_custom tl t
      end
```

This combination—structural defaults with typed escape hatches—has proved more useful than attempting to make the automatic policy cover every domain type.

6.7 Graphical User Interfaces

One of the most visible applications of runtime types is the automatic generation of user interfaces. For example, the following record type can be understood to specify a UI form consisting of a date picker, a dropdown menu with two options, a numeric input, a text input, and a currency selector.

Figure 2 shows the native Windows UI and a similar web UI generated from this definition.

Table 1. Representative default GUI interpretations.

Type element	GUI element
`string`	Text field
`bool`	Checkbox
Constant sum type	Combo box
Record	Form containing one control per field
List of records	Grid
`'a option`	Checkbox controlling an editor for `'a`
Nominal abstract type	Domain-specific widget
`[@t labeled = "..."]`	Custom label for a control or group
`[@t folded]`	Collapsible nested form
`[@t few_digits = "N"]`	Numeric editor with a given decimal representation
`[@t read_only]`	Read-only control
`[@t init = "..."]`	Initial value for a control (in variant syntax)

These two renderers were developed for different UI technologies, but the application types did not have to change. Structural information supplies a usable default form, type properties customize individual occurrences, and registries provide widgets for nominal abstract types such as dates, currencies, and financial underlyings. This division has been important in practice: domain types remain independent of a particular GUI toolkit, while a new renderer can reuse the conventions accumulated by earlier ones. Runtime types do not eliminate UI-specific work, but they concentrate it in reusable interpreters instead of distributing it across application forms.

6.8 Command-Line Interfaces

The same interpretation applies to function types. The `Mlfi_arg` library derives a command-line interface from a unit-returning function. Its main default interpretations are summarized below.

Table 2. Representative default command-line interpretations.

Type element	Command-line element
Constructor	Command or subcommand
Constructor argument	Arguments of the corresponding command
Labelled argument	Named option
Unlabelled function argument	Positional argument
Optional argument	Optional argument
`'a option`	Optional argument
`list`	Repeated argument
`bool`	Flag

Type properties supply documentation and validation metadata and can refine these defaults.

```
type command =
  | Cat of { filename : string }
  | Echo of string

let main = function
  | Cat { filename } -> Printf.printf "cat: %s\n" filename
  | Echo text -> print_endline text
```

```

let () = Mlfi_arg.eval "example" main
```

6.9 Database Schemas

Database tables are also described by runtime types. When a table is created, its witness is traversed to obtain the stored record's fields, their types, and database-related properties. This produces the logical column description used by the database layer. The table retains the witness, so later operations can convert between typed records and the untyped representation used by backup, restore, and migration code.

Table 3. Representative default database interpretations.

Type element	Database element
Record type	Table row and its logical schema
Record field	Column
`int`, `string`, ...	Corresponding SQL scalar type
`'a option`	Nullable column
Lazy field	Column fetched on demand
`'a dblink`	Foreign-key column referring to a row of type `'a`
`[@t indexed]`	SQL index on the corresponding column
`[@t unique]`	Uniqueness constraint on the corresponding column

```
type 'a table
type 'a dblink

type 'a dbobj = private {
  dblink: 'a dblink;
  dbversion: int64;
  dbval: 'a;
}
```

Here, `'a table` is a handle for a database table storing records of type `'a`, `'a dblink` represents a stable database identity of an `'a` object, and `'a dbobj` represents the actual value plus identity and a version number used to keep track of concurrent updates. The essential CRUD operations preserve the stored record type throughout:

```
val create: 'a table -> 'a -> 'a dbobj
val follow: 'a table -> 'a dblink -> 'a dbobj option
val update: 'a table -> 'a dbobj -> 'a -> [ `Ok of 'a dbobj | `Modified | `Error of string ]
val delete: 'a table -> 'a dbobj -> [ `Ok | `Modified | `Error of string ]
```

Updates use the version carried by `dbobj` for optimistic concurrency control; `` `Modified `` reports that another writer changed the object after it was read.

Schema construction:

```
val create_table: t:'a ttype -> string -> 'a table
```

Normal usage is simply:

```
type person = {
  name: string [@t indexed; unique];
  age: int;
```

```
  active: bool;
  nickname: string option;
}

let people = create_table ~t:[%t: person] "people"
```

The mapping is natural: optional fields become nullable columns; the `[@t indexed]` property generates a SQL index; `[@t unique]` generates a unique constraint; primitive types receive native SQL types; and records, variants, lists, and other nonprimitive values are "variantized" and stored in a binary or text column. A `dblink` becomes an integer column and can generate a foreign-key constraint, while `Lazy.t` marks a column that is omitted from the initial query and fetched on demand.

Runtime types continue to be useful after schema creation. SQL `WHERE` clauses are represented by typed conditions built from type paths:

```
type 'a condition

val cond_any: 'a condition
val cond_and: 'a condition list -> 'a condition
val cond_field_equal: t:'b ttype -> ('a, 'b) type_path -> 'b -> 'a condition

val fetch_cond: 'a table -> 'a condition -> 'a dbobj list
```

For example:

```
let alices_aged_18 =
  fetch_cond people
    (cond_and [cond_field_equal [%p name] "Alice"; cond_field_equal [%p age] 18])
```

Here `[%p name]` ensures that the compared value is a string and `[%p age]` ensures that it is an integer. The same table witness is also retained for backup, restore, and migration code, where records must cross an untyped representation. The practical benefit is not merely avoiding a schema declaration: schema construction, queries, persistence, and migrations all agree on one typed description of the stored value.

6.10 Type-safe Marshalling

The marshalling layer stores a digest of the runtime type together with the serialized value. The digest is checked when reading back the value, which allows verifying whether the payload matches the expected type. Here, type safety means that a marshalled value is returned only when its recorded type matches the expected type; the handling of untrusted payloads is not addressed here.

```
val marshal: t:'a ttype -> 'a -> string
val unmarshal: t:'a ttype -> string -> 'a option
```

The digest is computed from a canonicalized, serialized form of the corresponding `'a ttype`. Morally, the functions are implemented as follows (this is just for illustration purposes; the actual implementation is more complex):

```
let digest_ttype t =
  Digest.string (Marshal.to_string t [])

let marshal ~t x =
  Marshal.to_string (digest_ttype t, x) []

```

```
let unmarshal ~t s =
  let digest, x = Marshal.from_string s 0 in
  if Digest.equal (digest_ttype t) digest then Some x else None
```

There is a limitation around nominal abstract types: as the runtime type only records an abstract type's name and parameters, changing its hidden representation will not change the digest. The type digest alone does not characterize the implementation of abstract types. In our codebase, we apply a preprocessing step so that abstract types are required to have custom marshalling behaviour attached to them.

7 RELATED APPROACHES

Runtime type witnesses are not unique to our system. Below we discuss PPX derivers and libraries which expose runtime representations of types together with associated operations.

Generally speaking, one key advantage of our approach is its integration with the type checker, which removes two sources of friction: the need to construct type witnesses by hand, and the need to identify the relevant type at call sites. Library-based approaches normally require the explicit construction of type representations. PPX derivers can produce concise calls, but the selected operation typically still identifies the type declaration, for example through a name such as `show_trade` or `Trade.show`. In our system, the same polymorphic generic operation can simply be written:

```
print value
```

The compiler supplies the runtime type from the statically inferred type of `value`. Neither a runtime-type argument nor a type-specific operation name clutters the call site, and the call need not be adapted when the inferred type changes.

7.1 Scrap Your Boilerplate and Multi-Stage Programming

A further family of techniques is inspired by Scrap Your Boilerplate (SYB) [9, 11]. These approaches organize generic programming around uniform ways to decompose and traverse values, with type-specific cases embedded in otherwise generic traversal schemes. Clients can define new generic operations, but do so against these traversal and decomposition interfaces rather than by inspecting a single structural type representation such as `xtype`. Straightforward implementations can have substantial interpretive overhead. This overhead can be mitigated by staging [9].

More generally, multi-stage systems such as MetaOCaml and, more recently, MacoCaml [10], are particularly relevant. These developments open new and exciting possibilities. Neither approach is part of standard OCaml, and we do not have enough expertise or practical experience with them to offer a useful comparison here.

7.2 PPX Derivers

PPX derivers [12] are probably the most common approach to generic programming in OCaml. A PPX deriver generates code at compile time from a type declaration. The resulting code is then type-checked and compiled as ordinary OCaml code.

A PPX deriver inspects a type declaration and generates a specialized operation such as `show_trade` or `json_of_trade`. This is attractive when a small, fixed set of operations is known in advance, and the generated code can be optimized for the particular declaration. Adding a genuinely new generic operation, however, usually means implementing another deriver. This is a substantially heavier endeavour than writing an ordinary recursive function over `xtype`.

The deriver is a purely syntactic part of the code-generation pipeline, normally through `ppxlib`. Although `ppxlib` absorbs much version-specific churn, the dependency on an implementation detail

of the compiler (the AST definition, `Parsetree`) is a source of fragility. The fact that PPXs do their work in a different stage than the rest of the compilation is another source of friction.

On the upside, a deriver can generate code specialized to the type declaration, with no need to interpret a type descriptor at runtime. At the same time, intensive use of derivers can increase generated-code size and compilation and linking times.

7.3 Runtime Types as a Library

Several OCaml libraries provide first-class representations of types. They share the idea of associating an OCaml type `'a` with a value describing it, but they make different choices about what is represented and what clients may do with the representation.

Among such libraries, we may mention Repr [14], Typegist [13], and Refl [15]. These three libraries are only a sample; other OCaml libraries for generic programming and type reflection exist. We briefly compare them along four main axes: whether clients can define new generic operations; how abstract types and their custom behaviour are handled; whether metadata can be attached to guide generic operations; and whether the representation describes a closed type or an open type expression.

The comparison is based only on their documentation and source code, rather than on actual use, so it may contain inaccuracies. We welcome corrections for future versions of this paper.

Repr. Repr [14] is a combinator library whose central type `'a Repr.t` represents values of the closed OCaml type `'a`. Combinators are provided to build up type witnesses: a representation of `'a list` is built by applying the `list` combinator to a representation of `'a`. Record, variant, and recursive representations can be assembled with similar combinators. The optional `ppx_repr` package can generate these representations from type declarations.

The type `'a Repr.t` is abstract. Repr interprets it into the family of fixed operations exposed by the library, which include pretty-printers, equality and comparison operations, hashing, and various codecs. A client can customize those operations and can map an abstract type to a public representation, but it is not possible to define an unrelated generic operation on the representation itself.

Repr's treatment of abstract types is similarly limited: an abstract representation can specify custom behaviour only for Repr's supported operations. This is convenient and explicit, but the customization is tied to that particular `Repr.t` value and to the operations anticipated by the library. Repr does not expose an open metadata mechanism analogous to type properties.

Typegist. Of the three libraries considered here, Typegist [13] is probably the closest to our system in terms of its user-facing API. It exposes a structural GADT, so clients can pattern match on a gist and define new generic functions. Like Repr, it describes a closed OCaml type.

Its most distinctive feature for this comparison is extensible, type-indexed metadata. Metadata can occur throughout a gist—on types, fields, constructors, and public representations—and generic functions may define their own typed metadata keys. This addresses the same need as type properties but with a different tradeoff. Typegist metadata is typed and extensible as a library mechanism. In contrast, the type properties of Section 5.1 are string-valued and less disciplined, but are propagated through unification.

Typegist deliberately permits more than one gist for the same OCaml type. An abstract type may expose no representation or several public representations, each with conversions to and from the abstract value. This is well suited to schemas and evolving external formats. It also means that a gist is a chosen structural view rather than a canonical type identity: equality of gists does not in general justify equality of the underlying OCaml types.

Refl. Refl [6, 15] is perhaps the most ambitious of the three libraries. It exposes a rich structural GADT indexed by nine parameters (!), tracking the described value, the form and arity of the description, recursive and GADT environments, and the positive and negative occurrences of parameters. It contains explicit type variables and applications. Type constructors such as `'a list` can be represented by themselves, rather than only their closed instances. Refl can also describe GADTs, which our deliberately smaller representation rejects.

The description is public, so new generic functionality can be defined by pattern matching. Since Refl can describe open type expressions, consumers can be parameterized by operations for their free type variables. These parameter environments are tracked in the type of the description, providing stronger static guarantees at the cost of more involved consumer APIs. Refl also supports extensible typed attributes which can be used to customize behaviour.

Refl does not support fully abstract types: its nominal witness occurs only in a named node that also contains the structural description of the type. Its core representation has no abstract-type node, and its PPX cannot derive a representation from a fully abstract declaration. It therefore provides neither a counterpart to our nominal abstract-type representation nor a global registration mechanism for attaching generic behaviour to such types.

Refl uses a PPX to simplify building type witnesses. Once that description has been derived, operations such as printing, comparison, mapping, folding, and construction are ordinary interpretations of it rather than separate derivers.

8 CONCLUSION

We have presented LexiFi's system of "runtime types," a pragmatic extension of OCaml developed over the years in the context of a codebase that relies heavily on generic infrastructure. The core elements are: a type for typed witnesses, a GADT view that exposes the shape of those witnesses in a type-safe manner, compiler support for building and passing witnesses, plus a mechanism for attaching string-valued metadata to types and ways to register custom behaviour for nominal abstract types. The compiler integration is the key that makes this programming model lightweight enough for daily use across an industrial codebase. For scale, an August 2026 snapshot contains 5,226 uses of `[%t: ...]` in 561 OCaml files, 9,925 type-property annotations in 681 files, and 3,928 uses of `[%p ...]` in 270 files.

Several lessons emerge from this experience. Compiler support for constructing and passing witnesses has been essential to making generic functions unobtrusive enough for pervasive use. Structural interpretation works best when combined with type-indexed escape hatches for abstract and domain-specific types.

The system is not perfect, and we have already mentioned several limitations throughout the report.

The system reflects the subset of OCaml that our generic infrastructure has needed. It supports primitive types, products, records, sums, lists, arrays, options, functions, objects, lazy values, and abstract types, but not GADTs or first-class modules.

Abstract types can only be represented when they have a global name. In particular, abstract types introduced by functor arguments or first-class modules cannot be represented. For the abstract types which are supported, the global name provides a nominal identity usable to register custom behaviour at those types. Functors (e.g., `Set` and `Map`) are also awkward to support generically (even if particular instances can be supported in a straightforward manner).

Type properties are string-valued. This makes them easy for the compiler to handle and for unrelated libraries to consume, but provides no static check that a property name or value is valid. Misspelled or conflicting properties cannot be detected statically.

Interpreting a runtime type descriptor carries a certain overhead. Whenever this overhead was too high for a particular use case, manually staging the generic function—that is, interpreting the type once to produce a specialized closure—was sufficient in our experience. Staging techniques as in [9] and [10] would likely be relevant in the future.

8.1 Implementation and Maintenance

Maintaining a compiler fork carries a cost. That said, maintenance of the patch has proved relatively easy. The part that generates type witnesses is small: the `[%t: _]` operator is translated to Lambda code that the programmer could, for the most part, have generated by hand. It is therefore independent of changes to the compiler after the Lambda representation, including changes to the middle end and the native-code back end, and it works transparently with other back ends, such as `js_of_ocaml`.

The `[%t: T]` operator generates type witnesses at compilation time. Their serialized representations are stored as data in the compilation unit, then deserialized and made available to the program at runtime.

The trickiest part of the patch is the treatment of type properties in the type checker, in particular their interaction with type expansion. This code is necessarily more closely tied to the low-level workings of the type checker.

In order to preserve ABI compatibility with the standard OCaml compiler, we perform a number of representation tricks to encode the new constructs (the `[%t: T]` operator, type properties and type paths) in terms of existing constructs (which are unlikely or impossible for the user to write). These "exotic" constructs are then intercepted after typechecking and eliminated before reaching Lambda generation.

Usually, when a new version of the compiler is released, the changes needed to the runtime type machinery itself are minimal or none. Sometimes the "type properties" part of the patch requires some care, but not always. The overall time needed to adapt our patch is small relative to the time needed to adapt the rest of the codebase due to changes in the standard library, new language features, changes to `compiler-libs`, bugs, etc. A tiresome part of the work is porting the patch to new versions of Merlin, with its modified type checker, as this requires carefully porting changes across two different Git trees. Upstreaming more of Merlin's type checker to the compiler will surely help in this respect.

Despite these limitations and maintenance costs, runtime types have provided a stable foundation for generic infrastructure across our codebase. Our experience suggests that modest compiler support for constructing and propagating type witnesses can make type-directed programming practical at industrial scale.

ACKNOWLEDGMENTS

I thank Jean-Marc Eber and Alain Frisch for devising many of the elements of the system, my colleagues at LexiFi for many fascinating discussions, and the anonymous reviewers of the presentation proposal for their helpful suggestions.

REFERENCES

[1] Alain Frisch. *Runtime types in OCaml.* https://www.lexifi.com/blog/ocaml/runtime-types/.
[2] Alain Frisch. *Dynamic types.* https://www.lexifi.com/blog/ocaml/dynamic-types/.
[3] Grégoire Henry and Jacques Garrigue. *Runtime types in OCaml.* https://ocaml.org/conferences/ocaml/2013/proposals/runtime-types.pdf.
[4] Grégoire Henry and Jacques Garrigue. *Runtime types in OCaml (Slides).* https://ocaml.org/conferences/ocaml/2013/slides/henry.pdf.
[5] Patrik Keller and Marc Lasson. *LexiFi Runtime Types.* https://watch.ocaml.org/w/rfAAeQ2xNjP7vwMQ1tr8ne.

[6] Thierry Martinez. *Type-safe type reflection for OCaml.* https://cambium.inria.fr/seminaires/transparents/20201109.Thierry.Martinez.pdf.
[7] Olivier Nicole. *Implicit source positions.* https://github.com/ocaml/ocaml/pull/13886.
[8] Florent Balestrieri and Michel Mauny. *Generic Programming in OCaml.* https://inria.hal.science/hal-01664286/.
[9] Jeremy Yallop. *Staged Generic Programming.* https://www.cl.cam.ac.uk/~jdy22/papers/staged-generic-programming.pdf.
[10] Dmitrij Szamozvancev, Leo White, Ningning Xie, and Jeremy Yallop. *Safe and efficient generic functions with MacoCaml.* https://www.cl.cam.ac.uk/~jdy22/papers/safe-and-efficient-generic-functions-with-macocaml.pdf.
[11] David Kaloper Meršinjak. *TPF: Trivial/Tagless Polytypic Functions.* https://github.com/pqwy/tpf.
[12] Jane Street. *Ppxlib user manual.* https://ocaml-ppx.github.io/ppxlib/ppxlib/.
[13] Daniel Bünzli. *Typegist.* https://erratique.ch/software/typegist.
[14] Thomas Gazagnaire and Craig Ferguson. *Repr.* https://github.com/mirage/repr.
[15] Thierry Martinez. *Refl.* https://github.com/ocamllibs/refl.

CHANGELOG

2026-08-24 Initial version.