***WARNING, REJECTED by the JFP** but may still be useful*, 16 pages, 2024. © Daniel C. Bünzli 2024
doi:xx.xxxx/xxxxx Document style kept for now, because layout was tailored for it. revision 1

FUNCTIONAL PEARL

An alphabet for your data soups

DANIEL C. BÜNZLI [*]
Institute of Philosophy, University of Bern, Switzerland
(e-mail: daniel.buenzli@erratique.ch)

Abstract

Dealing with ubiquitous but poorly typed and structured data models like JSON in ML languages is unpleasant. But it doesn't have to be. We show how to define a generalized algebraic datatype whose values denote bidirectional maps between subsets of values of a data model and the ML values of your choice. With suitable combinators these maps are quick and pleasant to define in a declarative style. The result can be used by generic functions that decode, encode, query and update data soups with nicely typed values.

1 Introduction

Processing generic data models like JSON in ML languages is unpleasant. ML values can be converted to these data models with pickler combinators (Kennedy, 2004) or datatype-generic programming techniques (Gibbons, 2007). However, partially or fully modelling *their* data schemas remains cumbersome. Using a generic type for the data model works well in dynamically typed languages because it directly maps on their own type systems. But in ML this representation is unnatural and frustrating to use.

Instead, we show how to define a generalized algebraic datatype whose values denote bidirectional maps between subsets of values of the data model and the ML values you want to use. With appropriate combinators to construct them, these descriptions can be made quick to define in the *decoding* and *encoding* direction possibly eliding one if not immediately useful. The values of this datatype can be used by generic functions that:

- Directly decode or encode the data model to the ML values of your choice without constructing values of a generic representation of the data model.[1]
- Query and update data of partially modelled data schemas with arbitrary ML values.
- Automatically construct them from other datatype-generic representations you may already have defined for your ML types.

[*] Funded by the Swiss National Science Foundation (grant PP00P1_211010, *The Epistemology of Climate Change — Understanding the Climate Challenge*) and a grant from the OCaml Software Foundation.

[1] For JSON, since the shape of an object may depend on one of its members and that members are unordered, some form of generic buffering may be needed to decode certain data schemas.

Like pickler combinators (Kennedy, 2004), the definition of these values can be made pleasantly declarative. The decoding and encoding bureaucracy is left to the generic functions that interpret the datatype. One way of understanding the datatype is to devise pickler combinators for the data model — rather than for the values of the ML language – but leave out the specific value coding machinery open for interpretation. Another way is to see it as a tagged final coding of the data model. Concretely the datatype allows to interpose your own functions at each data model value decoding and encoding step. These functions can be lossy or creative which naturally leads to data queries and data updates.

In what follows we focus on presenting the datatype. Providing an ergonomic set of combinators to construct its values is important but less difficult. Definitions are kept as simple as possible to expose the essence of this *finally tagged* representation. A practical implementation[2] should enrich these definitions with documentation strings for data schema documentation generation, text locations for human friendly error reporting and text layout information for layout preserving updates.

We use OCAML (Leroy *et al.*, 2023) for the ML language and JSON (Bray, 2017) for the data model, but as we conclude in Section 6 with the recipe, this technique is independent of them. For conciseness we use exceptions to represent partial functions but signatures can be changed to use explicit `result` or `either` return types where needed. No effects are needed from the ML language.

2 A generic representation

First we define the type `Json.t`, a generic representation for JSON values in ML. In essence nothing more than an abstract syntax tree for JSON text with one case for each sort of value.

```
module Json = struct
  type t =
  | Null of unit | Bool of bool | Number of float | String of string
  | Array of t list | Obj of obj and obj = mem list and mem = string * t
end
```

As can be seen later, the type `Json.t` remains useful. However it is the type that is unacceptable to work with in ML. Given a fixed data schema to process we do not want to manipulate this soup of values:

- We want objects to be represented by proper record or variant types. Not by `Json.obj` association lists that must be dynamically name checked for expectations.
- We want to get typed values on array element and object member access. Not generic `Json.t` values that must be dynamically type checked for expectations.

3 A typed representation to interpret

To replace these generic values by the ML values we want, we introduce the type `'a jsont` whose values denote subsets of JSON values and their bidirectional map to ML values of type `'a`.

[2] One can be found at `https://erratique.ch/software/jsont`

We call these values “JSON types”. They belong to the following generalized algebraic datatype whose cases and elided definitions are detailed in subsequent sections. The Rec case is bureaucracy the laziest readers do not need to care about, it types recursives JSON values if your ML is strict.

```
type ('a, 'b) base_map = ...
type ('a, 'elt, 'builder) array_map = ...
and ('o, 'dec) obj_map = ...
and 'a any_map = ...
and ('a, 'b) map = ...
and _ jsont =
| Null : (unit, 'b) base_map -> 'b jsont
| Bool : (bool, 'b) base_map -> 'b jsont
| Number : (float, 'b) base_map -> 'b jsont
| String : (string, 'b) base_map -> 'b jsont
| Array : ('a, 'elt, 'builder) array_map -> 'a jsont
| Obj : ('o, 'o) obj_map -> 'o jsont
| Any : 'a any_map -> 'a jsont
| Map : ('a, 'b) map -> 'b jsont
| Rec : 'a jsont Lazy.t -> 'a jsont
```

Except for Any, Map and Rec, the cases of the type 'a jsont are in direct correspondence with those of Json.t. But rather than storing data in the cases we have functions to bidirectionally map them to values of a type 'a. The 'a jsont values are used alongside decoding and encoding processes to directly check and transform the shape of the data.

For instance we can implement (see Appendix and Section 4) these two functions which decode and encode generic Json.t values with ML values:

```
val decode : 'a jsont -> Json.t -> 'a
val encode : 'a jsont -> 'a -> Json.t
```

Representing JSON data with ML values becomes a matter of defining suitable 'a jsont values. For example this kind of JSON object for messages:

```
{ "content": "J'aime pas la soupe", "public": true }
```

can be represented in ML by a record with two fields. Using the record’s natural constructor and field accessors, combinators whose implementation is given in Section 3.5.3, and OCAML’s reverse function application operator |>, this kind of object is described by:

```
module Message = struct
  type t = { content : string; public : bool }
  let make content public = { content; public }
  let content msg = msg.content
  let public msg = msg.public
  let jsont : t jsont =
    obj_map make
    |> obj_mem "content" string ~enc:content
    |> obj_mem "public" bool ~enc:public
    |> obj_finish
end
```

3.1 Base cases

Every base case carries a value of type `base_map`:

```
type ('a, 'b) base_map =
{ dec : 'a -> 'b;
  enc : 'b -> 'a; }
```

Values of this type describe bidirectional maps from values of type `'a` to `'b`. They are used to transform the canonical ML type `'a` chosen for a JSON base type to the one we want to use. The base cases are as follows:

- `Null m` maps JSON nulls to type `'a` by mapping `unit` values with `m`.
- `Bool m` maps JSON booleans to type `'a` by mapping `bool` values with `m`.
- `Number m` maps JSON numbers or nulls[3] to type `'a` by mapping `float` values with `m`.
- `String m` maps unescaped JSON strings to type `'a` by mapping `string` values with `m`.

Most of the time the map `m` used with base cases is the identity map. But having maps on base types is part of the strategy to interpose functions in every coding context. This is particulary useful for JSON strings which are JSON's universal type: all sorts of enumerations, better represented by variants in ML, can be found in them. More amusing, to reliably interchange 64-bit integers with JSON you need to encode them in strings.[4]

3.2 Map case

The elided type `map` used by `Map` is:

```
and ('a, 'b) map =
{ dom : 'a jsont;
  map : ('a, 'b) base_map; }
```

A `Map m` value changes the ML type of the JSON type `m.dom` from `'a` to `'b`. It is a tool for composing `jsont` values. If the reader wonders whether it is not simpler to expose a base case like `String m` by the value `{dom = String; map = m}`, the answer is rather negative. It is not directly evident in our simpler exposition but having maps in base cases provides the proper coding context for erroring or text layout preserving. This context may be more difficult to recover or no longer be available to generic functions when they get to process the `Map` case which is not syntactically related to JSON text.

[3] The semantics of JSON numbers is left to be desired. Interoperable JSON implementations map JSON numbers to IEEE 754 `binary64` values. But they are *not* such values: NAN and infinities cannot be represented. As of writing, the most widely deployed and formally defined JSON encoder, namely ECMASCRIPT's `JSON.stringify` (Guo, 2023), lossily encodes any non-finite floating point value by `null`.

[4] Again, interoperable JSON implementations map JSON numbers to IEEE 754 `binary64` values. Hence the only integers that can be interchanged safely without precision loss are those in the range $[-2^{53}; 2^{53}]$.

3.3 Array case

The elided type `array_map` used by `Array` is:

```
and ('array, 'elt, 'builder) array_map =
{ elt : 'elt jsont;
  dec_empty : 'builder;
  dec_skip : 'builder -> int -> bool;
  dec_add : 'builder -> int -> 'elt -> 'builder;
  dec_finish : 'builder -> 'array;
  enc : 'acc. ('acc -> 'elt -> 'acc) -> 'acc -> 'array -> 'acc; }
```

An `Array m` value maps JSON arrays of uniform JSON type `m.elt` to values of type `'array` built using values of type `'builder`. The record `m` explains how to construct and deconstruct an `'array` value. For decoding, we start with the value `m.dec_empty`, the element in the JSON array at index `i` is added with `m.dec_add`, unless `m.dec_skip` is **`true`** on `i` (the purpose of `dec_skip` will become clear later), and the final array is returned by `m.dec_finish`. For encoding, the `m.enc` function folds over the elements of an `'array` value to encode them to the JSON array.

3.4 Any case

The elided type `any_map` used by `Any` is:

```
and 'a any_map =
{ dec_null : 'a jsont option;
  dec_bool : 'a jsont option;
  dec_number : 'a jsont option;
  dec_string : 'a jsont option;
  dec_array : 'a jsont option;
  dec_obj : 'a jsont option;
  enc : 'a -> 'a jsont; }
```

An `Any m` value maps sets of JSON values with multiple sorts to values of type `'a`. It embeds dynamic typing in our datatype. It also allows to decode and encode with different sorts of JSON values. For decoding a JSON value of sort `t`, a generic function uses the JSON type `m.dec_t` or errors if `None`. For encoding, the `m.enc` function returns the JSON type to use with the value.

Given a JSON type value `t` the following `option` combinator uses `Any` to make it nullable in JSON. The result of `option t` is a JSON type that maps JSON null values to `None` and otherwise maps JSON values as `t` does but with successful results wrapped by `Some`.

```
let option : 'a jsont -> 'a option jsont = fun t ->
  let none = Null { dec = Fun.const None; enc = Fun.const () } in
  let some = Map { dom = t; map = {dec = Option.some; enc = Option.get}}in
  let enc = function None -> none | Some _ -> some in
  let none = Some none and some = Some some in
  Any { dec_null = none; dec_bool = some; dec_number = some;
        dec_string = some; dec_array = some; dec_obj = some; enc; }
```

The `Any` case also allows to devise the JSON type `json` which maps any JSON value to its generic representation:

```
let json : Json.t jsont = ...
```

Its definition is left as an exercice for the reader but this value is a must for partial data schema modelling.

3.5 Obj case

Mapping objects is more involved and the design is less self-evident. Challenges are that members in JSON objects are unordered, that the shape of an object may depend on the value of one if its members[5] and that duplicate member names is undefined behaviour.[6] This means that we cannot rely on a fixed member ordering to construct the ML value of an object and worse, that we may have to wait for its last member to type check it.

To narrow the design space, we focus on a few patterns found in JSON data schemas that we want to support without fuss while retaining efficient decodes for object shapes that are known beforehand. These patterns are:

1. Objects as records. Member names and their types are known beforehand. Members are required or optional in which case they have a default value.
2. Objects as uniform key-value maps. Member names of the object are unknown but their values are all of the same type. This must compose with pattern 1. as with the JSON type `json` (Section 3.4) it enables partial object modelling and supports data schemas that allow foreign members in their objects.
3. Objects as sums. There is a distinguished *case member*, for example named `"type"`, `"class"` or `"version"`, and its value further determines an object shape described using pattern 1., 2. or 3.

Finally we want JSON object maps to be defined through functions that are already naturally provided for our ML types: constructors and accessors.

If the shape of an object cannot be captured by these patterns, it is always possible to map it to a uniform `Json.t` key-value map using pattern 2. followed by a `Map` to sort things out. This provides an ultimate escape hatch at the cost of unconditionnaly going through the generic representation.

3.5.1 (De)constructing arbitrary ML values for JSON objects

We want to represent JSON objects by arbitrary ML values of type `'o` which hold member values with their own distinct types `'a`$_1$, `'a`$_2$, etc.

For encoding this is easily tackled by having one projection function of type `'o -> 'a`$_i$ for each object member. For decoding we need to provide a constructor function with one argument per member value that returns a value of type `'o`. To manipulate this constructor we use a datatype morally equivalent to this representation of a function application:

[5] This is not a built-in mechanism of the data model but out-of-band constraints mandated by data schemas.
[6] But ECMASCRIPT's formally defined decoder `JSON.parse` (Guo, 2023) mandates "last one takes over."

```
type ('ret, 'f) app =
| Fun : 'f -> ('ret, 'f) app
| App : ('ret, 'a -> 'b) app * 'a -> ('ret, 'b) app
```

In a value of type `app` we can lift an arbitrary function `f` returning `'ret` with the `Fun` case and instrument each argument application with `App` cases until `f` is fully "applied" to a value of type `('ret, 'ret) app`. We store object constructors in a similar data type but since we do not have the argument values yet we use a type witness[7] to serve as a placeholder for the member value:

```
type ('ret, 'f) dec_fun =
| Dec_fun : 'f -> ('ret, 'f) dec_fun
| Dec_app : ('ret, 'a -> 'b) dec_fun * 'a Type.Id.t -> ('ret, 'b) dec_fun
```

This allows to decode unordered and individually typed member values as they come, store them by type witness in an heterogeneous dictionary `Dict.t` (see implementation in the Appendix) and, once we have collected all member values in the dictionary, we can invoke the constructor to get the ML value for the object with this function:

```
let rec apply_dict : type ret f. (ret, f) dec_fun -> Dict.t -> f =
fun dec dict -> match dec with
| Dec_fun f -> f
| Dec_app (f,arg) -> (apply_dict f dict) (Option.get (Dict.find arg dict))
```

For fully known object shapes this mechanism allows decoders to directly decode objects and their unordered member values to the representations we want to use in ML.

3.5.2 Object maps

The elided type `obj_map` used by `Obj` is:

```
and ('o, 'dec) obj_map =
{ dec : ('o, 'dec) dec_fun;
  mem_decs :  mem_dec String_map.t;
  mem_encs : 'o mem_enc list;
  shape : 'o obj_shape; }
```

An `Obj m` value maps a JSON object to a value of type `'o`. The `m.dec` field holds the constructor function for `'o` values. The `Obj` case in the definition of `jsont` (Section 3) constrains the `'dec` parameter to be equal to `'o` which ensures that the contructor is fully "applied". Remaining fields of the record are described in subsequent sections.

3.5.3 Member maps

The `m.mem_decs` and `m.mem_encs` fields of `obj_map` describe members of the object that are known beforehand. Both fields hold the same values of type `mem_map` but they are sorted differently and their type parameters are hidden in slighly different ways to accomodate decoding and encoding processes. These types are defined by:

[7] Available in the OCAML standard library in `Type.Id` since OCAML 5.1

```
and    mem_dec = Mem_dec : ('o, 'a) mem_map ->    mem_dec
and 'o mem_enc = Mem_enc : ('o, 'a) mem_map -> 'o mem_enc
and ('o, 'a) mem_map =
{ name : string;
  type' : 'a jsont;
  id : 'a Type.Id.t;
  dec_absent : 'a option;
  enc : 'o -> 'a;
  enc_omit : 'a -> bool; }
```

A value `mm` of type `mem_map` maps a member `'a` of a JSON object mapped to `'o`. `mm.name` is the member name. `mm.type'` is the JSON type of its value. `mm.id` is the type witness to represent the member value in the constructor of `'o`. `mm.dec_absent` is a value to use if the member is absent on decodes; `None` means error on absence. `mm.enc` is the function to get back the member value from `'o` for encoding. `mm.enc_omit` is a predicate on the value returned by `mm.enc` to decide whether it should be omitted on encoding; usually this tests for equality with the value mentioned in `mm.dec_absent`.

A member map `mm` needs to be added to an object map `m` in `m.mem_decs`, `m.mem_encs` and the `mm.id` type witness must be applied to the object constructor in `m.dec`. This is the duty of combinators. For example this one describes a required member and adds it to an object map:

```
let obj_mem :
  string -> 'a jsont -> enc:('o -> 'a) ->
  ('o, 'a -> 'b) obj_map -> ('o, 'b) obj_map
=
fun name type' ~enc obj_map ->
  let id = Type.Id.make () in
  let dec_absent = None and enc_omit = Fun.const false in
  let mm = { name; type'; id; dec_absent; enc; enc_omit } in
  let dec = Dec_app (obj_map.dec, mm.id) in
  let mem_decs = String_map.add mm.name (Mem_dec mm) obj_map.mem_decs in
  let mem_encs = Mem_enc mm :: obj_map.mem_encs in
  { obj_map with dec; mem_decs; mem_encs; }
```

At this point we can provide the full implementations of the combinators used in the message object modelling example given in Section 3.

```
let bool = Bool { dec = Fun.id; enc = Fun.id }
let string = String { dec = Fun.id; enc = Fun.id }
let obj_finish o = Obj { o with mem_encs = List.rev o.mem_encs }
let obj_map : 'dec -> ('o, 'dec) obj_map = fun make ->
  let dec = Dec_fun make and shape = Obj_basic Unknown_skip in
  { dec; mem_decs = String_map.empty; mem_encs = []; shape }
```

3.5.4 *Object shapes*

The last field of the `obj_map` type to describe is the `shape` field of type `obj_shape`:

```
and 'o obj_shape =
| Obj_basic : ('o, 'mems, 'builder) unknown_mems -> 'o obj_shape
| Obj_cases : ('o, 'cases, 'tag) obj_cases -> 'o obj_shape
```

This value indicates whether the members described in the object map are the final word on the shape of the object:

- `Obj_basic u` indicates that the object's members are fully known and the way to handle unknown member is described by `u`, see Section 3.5.5.
- `Obj_cases cases` indicates that there is a case member described in `cases`. Each case member value gives another `obj_map` value which further describe the object, see Section 3.5.6.

The `obj_shape` type definition turns object map values into a decision tree with `Obj_cases` nodes, branches labelled by case member values and with `Obj_basic` leaves. Each path in this tree describes a complete object whose members depend on case member values found in the data. We assume that the combinators constructing these values enforce the constraint that no member is defined twice in a path from the root to a leaf.

Note that once you get an `Obj_basic` shape, all data dependent shapes have been determined and members can be directly decoded to their type without buffering them.

3.5.5 *Unknown members*

The type `unknown_mems` used by `Obj_basic` shapes is:

```
and ('o, 'mems, 'builder) unknown_mems =
| Unknown_skip : ('o, unit, unit) unknown_mems
| Unknown_error : ('o, unit, unit) unknown_mems
| Unknown_keep :
    ('mems, 'a, 'builder) mems_map * ('o -> 'mems) ->
    ('o, 'mems) unknown_mems
```

A value `u` of type `unknown_mems` maps to `'mems` the unknown members of a JSON object mapped to `'o`. It respectively indicates to skip, error, or keep them. In the latter case the `Unknown_keep (m, enc)` value describes with `enc` how to get them back from `'o` for encoding and with `m`, how to map them to a value of type `'mems`. The values `enc` and `m` are kept separate because the type `'o` is bespoke while unknown member maps can be reused across object maps. The value `m` is of this type:

```
and ('mems, 'a, 'builder) mems_map =
{ mems_type : 'a jsont;
  id : 'mems Type.Id.t;
  dec_empty : 'builder;
  dec_add : string -> 'a -> 'builder -> 'builder;
  dec_finish : 'builder -> 'mems
  enc : 'acc. (string -> 'a -> 'acc -> 'acc) -> 'mems -> 'acc -> 'acc }
```

This record maps unknown members of uniform JSON type `m.mems_type` to a value of type `'mems` built using values of types `'builder`. Use the JSON type `json` (Section 3.4) in `m.mems_type` for partial object modelling or objects that need to preserve foreign members. `m.id` is the type witness to represent the `'mems` value in the object constructor. For decoding, we start with the value `m.dec_empty`, unknown members are added with `m.dec_add` and the final `'mems` value is returned by `m.dec_finish`. For encoding `m.enc` allows to recover from `'mems` the unknown members to encode them in the JSON object.

3.5.6 Object cases

Type type `obj_cases` used by `Obj_cases` shapes is:

```
and ('o, 'cases, 'tag) obj_cases =
{ tag : ('o, 'tag) mem_map; (* 'o is irrelevant, 'tag is not stored *)
  tag_compare : 'tag -> 'tag -> int;
  id : 'cases Type.Id.t;
  cases : ('cases, 'tag) case list;
  enc : 'o -> 'cases;
  enc_case : 'cases -> ('cases, 'tag) case_value; }
```

A value `m` of type `obj_cases` maps to `'cases` the object cases of an object mapped to `'o`. Cases are selected by the value of a case member of type `'tag` described in `m.tag`. Tag values are not stored in `'o` (the decoded case value is) so the `'o` parameter, `m.tag.id` and `m.tag.enc` are unused here. `m.tag_compare` allows to compare case tags. `m.id` is the type witness to represent the cases in the constructor of `'o`. `m.cases` is the list of cases. This is not a function on `'tag` values in order to make the description enumerable (e.g. for schema documentation generation). The type `case` hides the `'case` parameter of the type `case_map` which describes cases:

```
and ('cases, 'tag) case =
| Case : ('cases, 'case, 'tag) case_map -> ('cases, 'tag) case

and ('cases, 'case, 'tag) case_map =
{ tag : 'tag;
  obj_map : ('case, 'case) obj_map;
  dec : 'case -> 'cases; }
```

A value `cm` of type `case_map` describes a case of type `'case` part of the type `'cases`. `cm.tag` is the tag value that identifies the case in the data. `cm.obj_map` describes the additional shape this case gives to the object. `cm.dec` injects the decoded case into the type that gathers them.

For encoding cases, the `m.enc` function of `obj_cases` gets back the case from `'o`. To find out how to encode it, the function `m.enc_case` is used. It returns a value of type `case_value` which has a the actual case value and its map for encoding:

```
and ('cases, 'tag) case_value =
| Case_value :
    ('cases, 'case, 'tag) case_map * 'case -> ('cases, 'tag) case_value
```

The `m.enc_case` function is the only ad-hoc function that needs to be devised specifically for `jsont` values. All the other functions to describe objects are natural constructors and accessors of ML types.

The design for object cases allows to map them to a record type which has common fields for all cases and a field for the cases:

```
type type' = C1 of C1.t | C2 of C2.t ...
type t =
{ ... (* Fields common to all cases *); type': type';}
```

but they can also be described individually and mapped to a "toplevel" variant type if `'cases` coincides with `'o`:

```
type t = C1 of C1.t | C2 of C2.t ...
```

4 Decode and encode

Given a `jsont` value we can decode and encode JSON with ML values without constructing generic `Json.t` values; except transiently for decoding object instances with data dependent shapes and poorly ordered members. Implementing a JSON codec is beyond the scope of this paper but the Appendix has implementations for `decode` and `encode` functions that convert ML values with generic `Json.t` values.

For `decode` we took care not to assume full in-memory access to an object's members. It thus shows how a decoder can proceed to provide best-effort on-the-fly decoding. Except for case members, the last occurence of duplicate members takes over, however all definitions must type as defined by the object map otherwise the decode errors. These limitations on duplicate members could be lifted with a more complex decoder but it may not be worth the trouble. The case for objects is more intricate than we would like it to be, but we blame JSON's loose specification for that.

Otherwise the implementation of these functions mostly consists in recursing on the `jsont` values to boringly invoke the menagerie of functions that are packed therein.

5 Query and update

Since we can now interpose our functions in every coding context we get a very flexible data processing system. A type for data queries and a function to execute them can be as simple as:

```
type 'a query = 'a jsont
let query : 'a query -> Json.t -> 'a = decode
```

In this view, queries are just transforming decodes. Their encoding direction can be made to fail or defined with anything that feels sensitive to encode the query result to.

To navigate the structure of JSON values to apply a query on a subtree, the following composable indexing combinators can be used:

```
let get_mem : string -> 'a query -> 'a query = fun name q ->
  obj_map Fun.id |> obj_mem name q ~enc:Fun.id |> obj_finish
```

```
let get_nth : int -> 'a query -> 'a query = fun nth q ->
  let dec_empty = None and dec_add _ _ v = Some v in
  let dec_skip _ k = nth <> k in
  let dec_finish = function None -> failwith "too short" | Some v -> v in
  let enc f acc v = f acc v (* Singleton array with the query result *) in
  Array { elt = q; dec_empty; dec_add; dec_skip; dec_finish; enc }
```

The `get_nth` combinator explains the presence of `dec_skip` in the `array_map` type (Section 3.3). The query `q` only needs to succeed on the `nth` element. Without `dec_skip` we would apply it on every element of the array which is undesirable. The `dec_skip` field is the only bit in the design that was specifically added to support queries. For objects, skipping unknown members is quite natural to have in order to support data schema evolution.

Typed updates of JSON data is easy to specify as `Json.t` returning JSON types. Decoders invoking such queries return updated JSON as `Json.t` values. Here is a kernel of composable combinators to peform updates:

```
val update_mem : string -> 'a jsont -> Json.t jsont
val update_nth : int -> 'a jsont -> Json.t jsont
val delete_mem : string -> Json.t jsont
val delete_nth : int -> Json.t jsont
val const : 'a jsont -> 'a -> 'a jsont
```

The `update_mem` and `update_nth` combinators apply on the member or index value the decoder of the given JSON type and replace it with the encoding of the result. Chaining update combinators allows to navigate arbitrarily nested JSON to apply an update. All these combinators are simple `Map` over the JSON type `json` (Section 3.4) with suitable uses of `encode` and `decode`. The implementations of `update_mem`, `delete_mem` and `const` are:

```
let update_mem : string -> 'a jsont -> Json.t jsont = fun name q ->
  let dec = function
  | Json.Obj ms ->
      let update (n, v as m) =
        if n = name then (n, encode q (decode q v)) else m
      in
      Json.Obj (List.map update ms)
  | _ -> failwith "type error"
  in
  Map { dom = json; map = { dec; enc = Fun.id } }

let delete_mem : string -> Json.t jsont = fun name ->
  let dec = function
  | Json.Obj ms -> Json.Obj (List.filter (fun (n, _) -> n <> name) ms)
  | _ -> type_error ()
  in
  Map { dom = json; map = { dec; enc = Fun.id } }

let const : 'a jsont -> 'a -> 'a jsont = fun t v ->
  let dec _ = v and enc _ = encode t v in
  Map { dom = json; map = { dec; enc } }
```

6 The recipe

None of what was presented here is specific to the JSON data model. A datatype similar to `jsont` (Section 3) can be devised for any data model. The recipe is as follows.

- A base case is needed for every base type of the model. Having maps in these cases allows to accurately represent their coding contexts. (Section 3.1)
- An `Array`-like case is needed for mapping the model's type for arrays. (Section 3.3)
- An `Obj`-like case is needed for mapping the model's type for key-value maps or records. The ML ingredients here are: projection functions for encoding and, for decoding, a constructor function instrumented by a datatype representing function applications using type witnesses to indirectly refer to argument values. (Section 3.5)
- An `Any`-like case is needed if the model is dynamically typed. It is used to map implicit sums of the model's types to a uniform ML type. (Section 3.4)
- The `Map` case is needed for composing map values. (Section 3.2)
- The `Rec` case is needed in a strict ML for representing recursive values of the data model. (Section 3)

And with this we hope to have made your future data soups more edible in ML.

References

Bray, T., Ed. (2017) The JavaScript Object Notation (JSON) Data Interchange Format. *RFC 8259*. https://doi.org/10.17487/RFC8259

Guo, S., Ficarra M., Gibbons, K., Eds (2023) ECMAScript® 2023 Language Specification. ECMA-262. https://262.ecma-international.org/14.0/

Gibbons, J. (2007) Datatype-Generic Programming. *Lecture Notes in Computer Science*. vol 4719. https://doi.org/10.1007/978-3-540-76786-2_1

Kennedy, A. J., (2004) Pickler combinators. *Journal of Functional Programming*. 14(6), 727-739. https://doi.org/10.1017/S0956796804005209

Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Sivaramakrishnan, KC, & Vouillon, J. (2023) The OCaml system release 5.1. Documentation and user's manual. https://ocaml.org/manual

Appendix

OCAML 5.1 implementation of the `decode` and `encode` functions mentioned in Section 4.

```
module String_map = Map.Make (String)

(* Errors *)

let type_error () = failwith "type error"
let unexpected_member n = failwith ("Unexpected member " ^ n)
let missing_member n = failwith ("Missing member " ^ n)
let unknown_case_tag () = failwith "Unknown case tag"
```

```
(* Heterogeneous key-value maps *)

module Dict = struct
  module M = Map.Make (Int)
  type binding = B : 'a Type.Id.t * 'a -> binding
  type t = binding M.t
  let empty = M.empty
  let add k v m = M.add (Type.Id.uid k) (B (k, v)) m
  let find : type a. a Type.Id.t -> t -> a option =
  fun k m -> match M.find_opt (Type.Id.uid k) m with
  | None -> None
  | Some B (k', v) ->
      match Type.Id.provably_equal k k' with
      | Some Type.Equal -> Some v | None -> assert false
end

(* Decode *)

let rec decode : type a. a jsont -> Json.t -> a =
fun t j -> match t with
| Null map -> (match j with Json.Null v -> map.dec v | _ -> type_error ())
| Bool map -> (match j with Json.Bool b -> map.dec b | _ -> type_error ())
| Number map ->
    (match j with
    | Json.Number n -> map.dec n | Json.Null _ -> map.dec Float.nan
    | _ -> type_error ())
| String map -> (match j with Json.String s -> map.dec s | _ -> type_error ())
| Array map ->
    (match j with Json.Array vs -> decode_array map vs | j -> type_error ())
| Obj map ->
    (match j with Json.Obj mems -> decode_obj map mems | j -> type_error ())
| Map map -> map.map.dec (decode map.dom j)
| Any map -> decode_any t map j
| Rec t -> decode (Lazy.force t) j

and decode_array : type a e b. (a, e, b) array_map -> Json.t list -> a =
fun map vs ->
  let add (i, a) v =
    i + 1, (if map.dec_skip a i then a else map.dec_add a i (decode map.elt v))
  in
  map.dec_finish (snd (List.fold_left add (0, map.dec_empty) vs))

and decode_obj : type o. (o, o) obj_map -> Json.obj -> o =
fun map mems ->
  apply_dict map.dec @@
  decode_obj_map map String_map.empty String_map.empty Dict.empty mems

and decode_obj_map : type o.
  (o, o) obj_map -> mem_dec String_map.t -> mem_dec String_map.t -> Dict.t ->
  Json.obj -> Dict.t
=
fun map mem_miss mem_decs dict mems ->
  let u n _ _ = invalid_arg (n ^ "member defined twice") in
  let mem_miss = String_map.union u mem_miss map.mem_decs in
  let mem_decs = String_map.union u mem_decs map.mem_decs in
  match map.shape with
  | Obj_cases cases -> decode_obj_case cases mem_miss mem_decs dict [] mems
  | Obj_basic u ->
      match u with
      | Unknown_skip -> decode_obj_basic u () mem_miss mem_decs dict mems
      | Unknown_error -> decode_obj_basic u () mem_miss mem_decs dict mems
      | Unknown_keep (map, _) ->
          decode_obj_basic u map.dec_empty mem_miss mem_decs dict mems

and decode_obj_basic : type o map builder.
  (o, map, builder) unknown_mems -> builder -> mem_dec String_map.t ->
  mem_dec String_map.t -> Dict.t -> Json.obj -> Dict.t
=
fun u umap mem_miss mem_decs dict -> function
| [] ->
    let dict = match u with
    | Unknown_skip | Unknown_error -> dict
    | Unknown_keep (map, _) -> Dict.add map.id (map.dec_finish umap) dict
    in
    let add_default _ (Mem_dec m) dict = match m.dec_absent with
    | Some v -> Dict.add m.id v dict | None -> missing_member m.name
    in
    String_map.fold add_default mem_miss dict
| (n, v) :: mems ->
    match String_map.find_opt n mem_decs with
    | Some (Mem_dec m) ->
        let dict = Dict.add m.id (decode m.type' v) dict in
        let mem_miss = String_map.remove n mem_miss in
        decode_obj_basic u umap mem_miss mem_decs dict mems
    | None ->
        match u with
        | Unknown_skip -> decode_obj_basic u umap mem_miss mem_decs dict mems
        | Unknown_error -> unexpected_member n
        | Unknown_keep (map, _) ->
            let umap = map.dec_add n (decode map.mems_type v) umap in
            decode_obj_basic u umap mem_miss mem_decs dict mems

and decode_obj_case : type o cases tag.
  (o, cases, tag) obj_cases -> mem_dec String_map.t -> mem_dec String_map.t ->
  Dict.t -> Json.obj -> Json.obj -> Dict.t
=
fun cases mem_miss mem_decs dict delay mems ->
  let decode_case_tag tag =
    let eq_tag (Case c) = cases.tag_compare c.tag tag = 0 in
    match List.find_opt eq_tag cases.cases with
    | None -> unknown_case_tag ()
    | Some (Case case) ->
        let mems = List.rev_append delay mems in
        let dict = decode_obj_map case.obj_map mem_miss mem_decs dict mems in
        Dict.add cases.id (case.dec (apply_dict case.obj_map.dec dict)) dict
  in
  match mems with
  | [] ->
      (match cases.tag.dec_absent with
      | Some t -> decode_case_tag t | None -> missing_member cases.tag.name)
  | (n, v as mem) :: mems ->
      if n = cases.tag.name then decode_case_tag (decode cases.tag.type' v) else
      match String_map.find_opt n mem_decs with
      | None -> decode_obj_case cases mem_miss mem_decs dict (mem :: delay) mems
      | Some (Mem_dec m) ->
          let dict = Dict.add m.id (decode m.type' v) dict in
          let mem_miss = String_map.remove n mem_miss in
          decode_obj_case cases mem_miss mem_decs dict delay mems
```

```
and decode_any : type a. a jsont -> a any_map -> Json.t -> a =
fun t map j ->
  let dec t m j = match m with Some t -> decode t j | None -> type_error () in
  match j with
  | Json.Null _ -> dec t map.dec_null j
  | Json.Bool _ -> dec t map.dec_bool j
  | Json.Number _ -> dec t map.dec_number j
  | Json.String _ -> dec t map.dec_string j
  | Json.Array _ -> dec t map.dec_array j
  | Json.Obj _ -> dec t map.dec_obj j

(* Encode *)

let rec encode : type a. a jsont -> a -> Json.t =
fun t v -> match t with
| Null map -> Json.Null (map.enc v)
| Bool map -> Json.Bool (map.enc v)
| Number map -> Json.Number (map.enc v)
| String map -> Json.String (map.enc v)
| Array map ->
    let encode_elt a elt = (encode map.elt elt) :: a in
    Json.Array (List.rev (map.enc encode_elt [] v))
| Obj map -> Json.Obj (List.rev (encode_obj map v []))
| Any map -> encode (map.enc v) v
| Map map -> encode map.dom (map.map.enc v)
| Rec t -> encode (Lazy.force t) v

and encode_obj : type o. (o, o) obj_map -> o -> Json.obj -> Json.obj =
fun map o obj ->
  let encode_mem obj (Mem_enc map) =
    let v = map.enc o in
    if map.enc_omit v then obj else (map.name, encode map.type' v) :: obj
  in
  let obj = List.fold_left encode_mem obj map.mem_encs in
  match map.shape with
  | Obj_basic (Unknown_keep (map, enc)) ->
      let encode_mem n v obj = (n, encode map.mems_type v) :: obj in
      map.enc encode_mem (enc o) obj
  | Obj_basic _ -> obj
  | Obj_cases cases ->
      let Case_value (case, c) = cases.enc_case (cases.enc o) in
      let obj =
        if cases.tag.enc_omit case.tag then obj else
        (cases.tag.name, encode cases.tag.type' case.tag) :: obj
      in
      encode_obj case.obj_map c obj
```