jsont.ml

December 9, 2014

```
(*---------------------------------------------------------------------------
   Copyright (c) 2014 Daniel C. BÃijnzli. All rights reserved.
   Distributed under the BSD3 license, see license at the end of the file.
   %%NAME%% release %%VERSION%%
  ---------------------------------------------------------------------------*)

let str = Printf.sprintf

let err_conv_default msg = str "could not convert default value (%s)" msg

let err_objd_dup_mem k n =
  str "object description %s: duplicate description for member %s" k n

let err_objd_dup_anon k =
  str "object description %s: duplicate description for anonymous members" k

let err_objd_used k =
  str "object description %s: description already in use" k

let err_some_combinator =
  str "Jsont.some misuse: cannot be used to encode None"

let err_oid = str "object not described by description"
let err_mem_oid n = str "object not described by description of member %s" n
let err_anon_oid = str "object not described by description of anonymous member"
let err_anon_mem n = str "no anonymous member named %s" n

(* Universal values, see http://mlton.org/UniversalType *)

type univ = exn
let univ_create (type s) () =
  let module M = struct exception E of s end in
  (fun x -> M.E x), (function M.E x -> x | _ -> assert false)

(* Value codecs *)

type ('a, 'b) value_decoder = 'a -> [ `Ok of 'b | `Error of string ]
type ('b, 'a) value_encoder = 'b -> 'a
type ('a, 'b) value_codec = ('a, 'b) value_decoder * ('b, 'a) value_encoder

(* Value locations *)

type loc = (int * int) * (int * int)
type 'a def = loc * 'a

let invalid_loc = (-1, 0), (-1, 0)
let is_invalid_loc l = l = invalid_loc
let invalid_def v = invalid_loc, v

let loc_merge (s, _) (_, e) = (s, e)

(* JSON values *)

type nat_string = string
type soup = Jsonm.lexeme def list

module Id = struct        (* uids for object description and object members. *)
  type t = int
  let nil = -1
  let create = let count = ref nil in fun () -> incr count; !count
```

```
  let compare : int -> int -> int = Pervasives.compare
end

module Mset = Set.Make (Id)
module Mmap = Map.Make (Id)
module Smap = Map.Make (String)

type obj =                                (* The type for JSON object values. *)
  { obj_oid : int;                                  (* object description id. *)
    obj_mems : univ Mmap.t;                    (* maps member ids to values. *)
    obj_anons : univ Smap.t; }     (* maps anonymous member names to values. *)

let obj_empty oid =
  { obj_oid = oid; obj_mems = Mmap.empty; obj_anons = Smap.empty }

(* JSON codecs and value description types *)

type error =
  [ `Json_decoder of Jsont_codec.error
  | `Type of string * string
  | `Value_decoder of string
  | `Member of string * string * [ `Dup | `Miss | `Unknown ] ]

let error_to_string = function
| `Json_decoder e -> (Format.asprintf "%a" Jsonm.pp_error e)
| `Value_decoder e -> e
| `Type (fnd, exp) -> str "value has type %s but expected type %s" fnd exp
| `Member (o, m, e)  ->
    str "member %s of object kind %s %s" m o begin match e with
    | `Dup -> "appears more than once"
    | `Miss -> "is missing"
    | `Unknown -> "is unknown"
    end

type 'a decode = [ `Await | `Ok of 'a def | `Error of error def ]
and 'a decoder =
  { dec_loc : bool;              (* [true] if locations should be computed. *)
    dec_dups : [ `Skip | `Error ];             (* behaviour on dup members. *)
    dec_unknown : [ `Skip | `Error ];      (* behaviour on unknown members. *)
    dec : Jsonm.decoder;                                  (* Jsonm decoder. *)
    mutable dec_lex : [ `End | `Lexeme of Jsonm.lexeme ];   (* last decode. *)
    mutable dec_lex_loc : loc option; (* if None is in Jsonm.decoded_range. *)
    mutable dec_next :                                      (* next decode. *)
      'a decoder ->
      [ `End | `Lexeme of Jsonm.lexeme | `Await | `Error of Jsonm.error ];
    mutable dec_ctx : obj list;          (* currently encoded object stack. *)
    mutable dec_delayed : soup list;                    (* delayed lexemes. *)
    mutable dec_k : 'a decoder -> 'a decode; }     (* decoder kontinuation. *)

and encode = [ `Partial | `Ok ]
and encoder =
  { enc : Jsonm.encoder;                                  (* Jsonm encoder. *)
    mutable enc_ctx : obj list;          (* currently encoded object stack. *)
    mutable enc_k : encoder -> encode }            (* encoder kontinuation. *)

and 'a descr =                                    (* JSON value description. *)
  { default : 'a;                                         (* default value. *)
    decode :                                              (* value decoder. *)
      'b. 'a descr -> ('a def -> 'b decoder -> 'b decode) ->
          'b decoder -> 'b decode;
    encode :                                              (* value encoder. *)
      'a descr -> 'a -> (encoder -> encode) -> encoder -> encode; }
```

```
type 'a mem =                                    (* JSON member description. *)
  { mem_oid : int;                                 (* object description id. *)
    mem_id : int;                                            (* member id. *)
    mem_name : string;                                     (* member name. *)
    mem_to_univ : 'a def -> univ;     (* converts value to universal value. *)
    mem_of_univ : univ -> 'a def;   (* converts value from universal value. *)
    mem_dep : mem_exists option;             (* member dependency (if any). *)
    mem_opt :                                      (* optional behaviour. *)
      [ `Yes | `Yes_rem of ('a -> 'a -> bool) | `No ];
    mem_descr : 'a descr; }                    (* member value description. *)

and mem_exists = Me : 'a mem -> mem_exists       (* hides mem's parameter. *)

type 'a anon =                           (* Unknown JSON member description. *)
  { anon_oid : int;                                (* object description id. *)
    anon_to_univ : 'a def -> univ;    (* converts value to universal value. *)
    anon_of_univ : univ -> 'a def;  (* converts value from universal value. *)
    anon_default : (string * 'a) list;             (* default anon members. *)
    anon_descr : 'a descr; }                          (* value description. *)

type anon_exists = Ae : 'a anon -> anon_exists

type objd =                                      (* JSON object description. *)
  { objd_id : int;                                 (* object description id. *)
    objd_kind : string;               (* a name for the object description. *)
    mutable objd_used : bool;          (* [true] when description was used. *)
    mutable objd_mem_list :                   (* object member description. *)
      (string * mem_exists) list;
    mutable objd_mems :              (* object member description as a map. *)
      mem_exists Smap.t;
    mutable objd_anon : anon_exists option; }(* unknown member description. *)

(* Decode *)

let loc d = match d.dec_lex_loc with
| None -> Jsonm.decoded_range d.dec
| Some loc -> loc

let ret v k d = d.dec_k <- k; v
let err loc err k d = ret (`Error (loc, err)) k d

let err_end k d = k d  (* some error will already have been reported. *)
let err_value_decoder loc msg k d = err loc (`Value_decoder msg) k d
let err_json_decoder loc e k d = err loc (`Json_decoder e) k d
let err_mem_dup loc okind n k d = err loc (`Member (okind, n, `Dup)) k d
let err_mem_miss loc okind n k d = err loc (`Member (okind, n, `Miss)) k d
let err_mem_unknown loc okind n k d = err loc (`Member (okind, n, `Unknown)) k d
let err_type loc fnd exp k d =
  let fnd = match fnd with
  | `Null -> "null"
  | `Bool _ -> "bool"
  | `Float _ -> "float"
  | `String _ -> "string"
  | `As | `Ae -> "array"
  | `Os | `Oe -> "object"
  | `Name _ -> "member"
  in
  err loc (`Type (fnd, exp)) k d

let rec dec_next k d = match d.dec_delayed with
| [] :: _ -> d.dec_lex <- `End; k d
```

```
| ((loc, l) :: soup) :: soups ->
    d.dec_delayed <- soup :: soups;
    d.dec_lex <- `Lexeme l;
    d.dec_lex_loc <- Some loc;
    k d
| [] ->
  d.dec_lex_loc <- None;
  match d.dec_next d with
  | `Lexeme _ as l -> d.dec_lex <- l; k d
  | `Await -> ret `Await (dec_next k) d
  | `End -> d.dec_lex <- `End; k d
  | `Error e -> err_json_decoder (loc d) e (dec_next k) d

let k_default descr k loc = k (loc, descr.default)
let k_default_range descr k start loc = k (loc_merge start loc, descr.default)

let skip_value k d =
  let start = loc d in
  let rec skip_struct last count k d =
    if count <= 0 then k (loc_merge start last) d else
    match d.dec_lex with
    | `Lexeme (`As | `Os) -> dec_next (skip_struct (loc d) (count + 1) k) d
    | `Lexeme (`Ae | `Oe) -> dec_next (skip_struct (loc d) (count - 1) k) d
    | `Lexeme _ -> dec_next (skip_struct (loc d) count k) d
    | `End -> err_end (k (loc_merge start (loc d))) d
  in
  match d.dec_lex with
  | `End | `Lexeme (`Null | `Bool _ | `Float _ | `String _ | `Ae | `Oe) ->
      dec_next (skip_struct (loc d) 0 k) d
  | `Lexeme (`Os | `As | `Name _) ->
      dec_next (skip_struct (loc d) 1 k) d

let rec finish v d = match d.dec_lex with
| `End -> d.dec_k <- (fun _ -> `Ok v); `Ok v
| `Lexeme l -> dec_next (finish v) d (* an error must have been reported. *)

let decoder ?(loc = false) ?(dups = `Skip) ?(unknown = `Skip) dec descr =
  let dec_k = dec_next (descr.decode descr (fun v -> dec_next (finish v))) in
  let dec_next d = Jsonm.decode d.dec in
  { dec_loc = loc; dec_dups = dups; dec_unknown = unknown;
    dec; dec_lex = (`Lexeme `Oe) (* dummy *); dec_lex_loc = None; dec_next;
    dec_delayed = []; dec_ctx = []; dec_k; }

let decode d = d.dec_k d
let decoder_decoder d = d.dec

(* Encode *)

let rec enc_next l k e = match Jsonm.encode e.enc l with
| `Ok -> k e
| `Partial -> e.enc_k <- (enc_next `Await k); `Partial

let encoder enc descr v =
  { enc; enc_ctx = [];
    enc_k = descr.encode descr v (enc_next `End (fun _ -> `Ok))  }

let encode e = e.enc_k e
let encoder_encoder e = e.enc

(* JSON value description combinators *)

let default d = d.default
```

```
let with_default v d = { d with default = v }

let decode_err typ descr k d = match d.dec_lex with
| `Lexeme l -> err_type (loc d) l typ (skip_value (k_default descr k)) d
| `End -> err_end (k_default descr k (loc d)) d

let bool =
  let decode descr k d = match d.dec_lex with
  | `Lexeme (`Bool b) -> dec_next (k (loc d, b)) d
  | _ -> decode_err "bool" descr k d
  in
  let encode descr b k e = enc_next (`Lexeme (`Bool b)) k e in
  { default = false; decode; encode }

let float =
  let decode descr k d = match d.dec_lex with
  | `Lexeme (`Float f) -> dec_next (k (loc d, f)) d
  | _ -> decode_err "float" descr k d
  in
  let encode descr f k e = enc_next (`Lexeme (`Float f)) k e in
  { default = 0.0; decode; encode }

let int =
  let decode descr k d = match d.dec_lex with
  | `Lexeme (`Float f) -> dec_next (k (loc d, int_of_float f)) d
  | _ -> decode_err "int" descr k d
  in
  let encode descr i k e = enc_next (`Lexeme (`Float (float_of_int i))) k e in
  { default = 0; decode; encode }

let int_strict =
  let decode descr k d = match d.dec_lex with
  | `Lexeme (`Float f as l) ->
      if f -. (floor f) <> 0.
      then err_type (loc d) l "int" (skip_value (k_default descr k)) d
      else dec_next (k (loc d, int_of_float f)) d
  | _ -> decode_err "int" descr k d
  in
  let encode descr i k e = enc_next (`Lexeme (`Float (float_of_int i))) k e in
  { default = 0; decode; encode }

let string =
  let decode descr k d = match d.dec_lex with
  | `Lexeme (`String s) -> dec_next (k (loc d, s)) d
  | _ -> decode_err "string" descr k d
  in
  let encode descr s k e = enc_next (`Lexeme (`String s)) k e in
  { default = ""; decode; encode }

let nat_string = string

let nullable base =
  let decode descr k d = match d.dec_lex with
  | `Lexeme `Null -> dec_next (k (loc d, None)) d
  | `Lexeme _ -> base.decode base (fun (loc, v) -> k (loc, Some v)) d
  | `End -> err_end (k_default descr k (loc d)) d
  in
  let encode descr v k e = match v with
  | None -> enc_next (`Lexeme `Null) k e
  | Some v -> base.encode base v k e
  in
  { default = Some base.default; decode; encode }
```

```
let codec ?default (vdec, venc) base =
  let default = match default with
  | Some v -> v
  | None ->
      match vdec base.default with
      | `Ok d -> d | `Error msg -> invalid_arg (err_conv_default msg)
  in
  let decode descr k d =
    let vdec k (loc, v) d = match vdec v with
    | `Error msg -> err_value_decoder loc msg (k_default descr k loc) d
    | `Ok v -> k (loc, v) d
    in
    base.decode base (vdec k) d
  in
  let encode descr v k e = base.encode base (venc v) k e in
  { default; decode; encode }

let type_match ~default decd encd =
  let decode descr k d = match d.dec_lex with
  | `Lexeme l ->
      let use typ = match decd typ with
      | `Ok vd -> vd.decode vd k d
      | `Error msg ->
          err_value_decoder (loc d) msg (skip_value (k_default descr k)) d
      in
      begin match l with
      | `Null -> use `Null | `Bool _ -> use `Bool | `Float _ -> use `Float
      | `String _ -> use `String
      | `As | `Ae (* array dec. will error *) -> use `Array
      | `Os | `Oe (* object dec will error *) | `Name _ -> use `Object
      end
  | `End -> err_end (k_default descr k (loc d)) d
  in
  let encode descr v k e = let descr = encd v in descr.encode descr v k e in
  { default; decode; encode }

let decode_soup descr k d =
  let start = loc d in
  let rec slurp last acc count k d =
    if count <= 0 then k (loc_merge start last, (List.rev acc)) d else
    let last = loc d in
    match d.dec_lex with
    | `Lexeme (`As | `Os as l) ->
        dec_next (slurp last ((last, l) :: acc) (count + 1) k) d
    | `Lexeme (`Ae | `Oe as l) ->
        dec_next (slurp last ((last, l) :: acc) (count - 1) k) d
    | `Lexeme l ->
        dec_next (slurp last ((last, l) :: acc) count k) d
    | `End ->
        err_end (k_default descr k (loc_merge start (loc d))) d
  in
  match d.dec_lex with
  | `Lexeme (`Null | `Bool _ | `Float _ | `String _ as l) ->
      dec_next (slurp start [start, l] 0 k) d
  | `Lexeme (`Os | `As) ->
      dec_next (slurp start [] 1 k) d
  | `Lexeme _ ->
      skip_value (k_default descr k) d (* error already reported. *)
  | `End ->
      err_end (k_default descr k start) d
```

```
let encode_soup descr soup k e =
  let rec vomit soup k e = match soup with
  | (_, l) :: acc -> enc_next (`Lexeme l) (vomit acc k) e
  | [] -> k e
  in
  vomit soup k e

let soup =
  let decode = decode_soup in
  let encode = encode_soup in
  { default = [(invalid_def `Null)]; decode; encode }

let some base =
  let decode descr k d = base.decode base (fun (loc, v) -> k (loc, Some v)) d in
  let encode descr v k e = match v with
  | None -> invalid_arg err_some_combinator
  | Some v -> base.encode base v k e
  in
  { default = None; decode; encode }

(* JSON array descriptions *)

let decode_array elt descr k d =
  let start = loc d in
  let rec loop acc k d = match d.dec_lex with
  | `Lexeme `Ae -> dec_next (k (loc_merge start (loc d), List.rev acc)) d
  | `End -> err_end (k_default descr k (loc_merge start (loc d))) d
  | _ -> elt.decode elt (fun (_, v) -> loop (v :: acc) k) d
  in
  match d.dec_lex with
  | `Lexeme `As -> dec_next (loop [] k) d
  |  _ -> decode_err "array" descr k d

let encode_array elt descr vs k e =
  let rec loop vs k e = match vs with
  | v :: vs -> elt.encode elt v (loop vs k) e
  | [] -> k e
  in
  enc_next (`Lexeme `As) (loop vs (enc_next (`Lexeme `Ae) k)) e

let array elt =
  let decode descr k d = decode_array elt descr k d in
  let encode descr k e = encode_array elt descr k e in
  { default = []; decode; encode }

let array_array elt =
  let c = (fun v -> `Ok (Array.of_list v)), (fun v -> Array.to_list v) in
  codec c (array elt)

(* JSON object descriptions *)

let objd ?kind () =
  let objd_id = Id.create () in
  let objd_kind = match kind with None -> str "o%d" objd_id | Some k -> k in
  let objd_used = false in
  let objd_mem_list = [] in
  let objd_mems = Smap.empty in
  let objd_anon = None in
  { objd_id; objd_kind; objd_used; objd_mem_list; objd_mems; objd_anon; }

let check_add objd name =
  if objd.objd_used then invalid_arg (err_objd_used objd.objd_kind) else
```

```
  if Smap.mem name objd.objd_mems
  then invalid_arg (err_objd_dup_mem objd.objd_kind name) else
  ()

let _mem ?(eq = ( = )) ?(opt = `No) mem_dep objd mem_name mem_descr =
  check_add objd mem_name;
  let mem_oid = objd.objd_id in
  let mem_id = Id.create () in
  let mem_to_univ, mem_of_univ = univ_create () in
  let mem_opt = match opt with `No | `Yes as v -> v | `Yes_rem -> `Yes_rem eq in
  let mem = { mem_oid; mem_id; mem_name; mem_to_univ; mem_of_univ;
              mem_dep; mem_opt; mem_descr }
  in
  let meme = Me mem in
  objd.objd_mem_list <- (mem_name, meme) :: objd.objd_mem_list;
  objd.objd_mems <- Smap.add mem_name meme objd.objd_mems;
  mem

let mem ?eq ?opt objd mem_name descr = _mem ?eq ?opt None objd mem_name descr

let mem_match ?eq ?opt objd mmatch name select =
  if objd.objd_id <>  mmatch.mem_oid
  then invalid_arg (err_mem_oid mmatch.mem_name) else
  let descr =
    let default = (select mmatch.mem_descr.default).default in
    let decode descr k d =
      let ctx = match d.dec_ctx with [] -> assert false | ctx :: _ -> ctx in
      let v = snd (mmatch.mem_of_univ (Mmap.find mmatch.mem_id ctx.obj_mems)) in
      let descr = select v in
      descr.decode descr k d
    in
    let encode descr memv k e =
      let ctx = match e.enc_ctx with [] -> assert false | ctx :: _ -> ctx in
      let v =
        try snd (mmatch.mem_of_univ (Mmap.find mmatch.mem_id ctx.obj_mems))
        with Not_found -> assert false
      in
      let descr = select v in
      descr.encode descr memv k e
    in
    { default; decode; encode }
  in
  _mem ?eq ?opt (Some (Me mmatch)) objd name descr

let anon ?default objd anon_descr =
  if objd.objd_used then invalid_arg (err_objd_used objd.objd_kind) else
  if objd.objd_anon <> None then invalid_arg (err_objd_dup_anon objd.objd_kind)
  else
  let anon_oid = objd.objd_id in
  let anon_to_univ, anon_of_univ = univ_create () in
  let anon_default = match default with None -> [] | Some v -> v in
  let a = { anon_oid; anon_to_univ; anon_of_univ; anon_default; anon_descr } in
  (objd.objd_anon <- Some (Ae a); a)

let objd_default objd =
  let obj_mems =
    let add_mem _ (Me m) acc =
      let v = m.mem_to_univ (invalid_def m.mem_descr.default) in
      Mmap.add m.mem_id v acc
    in
    Smap.fold add_mem objd.objd_mems Mmap.empty
  in
```

```
  let obj_anons = match objd.objd_anon with
  | None -> Smap.empty
  | Some (Ae a) ->
      let add_anon acc (k, v) =
        let v = a.anon_to_univ (invalid_def v) in
        Smap.add k v acc
      in
      List.fold_left add_anon Smap.empty a.anon_default
  in
  { obj_oid = objd.objd_id; obj_mems; obj_anons; }

let rec decode_miss_mems objd miss o k d = match miss with
| [] -> k o d
| (n, Me m) :: miss ->
    let v = m.mem_to_univ (invalid_def m.mem_descr.default) in
    let o = { o with obj_mems = Mmap.add m.mem_id v o.obj_mems } in
    match m.mem_opt with
    | `Yes | `Yes_rem _ -> decode_miss_mems objd miss o k d
    | `No ->
        err_mem_miss (loc d) objd.objd_kind n (decode_miss_mems objd miss o k) d

let decode_anon_mem objd miss o n loc k d = match objd.objd_anon with
| None ->
    begin match d.dec_unknown with
    | `Skip -> skip_value (fun _  -> k miss o) d
    | `Error ->
        err_mem_unknown loc objd.objd_kind n (skip_value (fun _ -> k miss o)) d
    end
| Some (Ae a) ->
    let add v d =
      let v = a.anon_to_univ v in
      k miss { o with obj_anons = Smap.add n v o.obj_anons } d
    in
    a.anon_descr.decode a.anon_descr add d

let rec decode_mem objd miss o n loc k d =
  match try Some (Smap.find n objd.objd_mems) with Not_found -> None with
  | None -> decode_anon_mem objd miss o n loc k d
  | Some (Me m) ->
      if Mmap.mem m.mem_id o.obj_mems && d.dec_dups = `Error then
        (* Second one takes over, report error and try again *)
        let o = { o with obj_mems = Mmap.remove m.mem_id o.obj_mems } in
        err_mem_dup loc objd.objd_kind n (decode_mem objd miss o n loc k) d
      else
      let add ~pop v d =
        let v = m.mem_to_univ v in
        let miss = Smap.remove n miss in
        if pop then d.dec_ctx <- List.tl d.dec_ctx;
        k miss { o with obj_mems = Mmap.add m.mem_id v o.obj_mems } d
      in
      match m.mem_dep with
      | None ->
          m.mem_descr.decode m.mem_descr (add ~pop:false) d
      | Some (Me dep) ->
          if Mmap.mem dep.mem_id o.obj_mems
          then begin
            d.dec_ctx <- o :: d.dec_ctx;
            m.mem_descr.decode m.mem_descr (add ~pop:true) d
          end
          else m.mem_descr.decode m.mem_descr (add ~pop:false) d

let decode_obj objd descr k d =
```

```
  let start = loc d in
  let rec loop k miss todo delayed o d = match d.dec_lex with
  | `Lexeme `Oe ->
      decode_miss_mems objd (Smap.bindings miss) o
        (fun o -> dec_next (k (loc_merge start (loc d), o))) d
  | `Lexeme (`Name n) ->
      dec_next (decode_mem objd miss o n (loc d) (loop k)) d
  | `Lexeme  l ->
      err_type (loc d) l "member or object end"
        (skip_value (k_default_range descr k start)) d
  | `End ->
      err_end (k_default descr k (loc_merge start (loc d))) d
  in
  match d.dec_lex with
  | `Lexeme `Os -> dec_next (loop k objd.objd_mems (obj_empty objd.objd_id)) d
  |  _ -> decode_err "object" descr k d

let rec encode_anons anond anons k e = match anons with
| [] -> k e
| (aname, avalue) :: anons ->
    match anond with
    | None -> assert false
    | Some (Ae a) ->
        let _, v = a.anon_of_univ avalue in
        enc_next (`Lexeme (`Name aname))
          (a.anon_descr.encode a.anon_descr v (encode_anons anond anons k)) e

let rec encode_mems memds mems k e = match memds with
| [] -> k e
| (n, Me m) :: memds ->
    let v = try Mmap.find m.mem_id mems with Not_found -> assert false in
    let _, v = m.mem_of_univ v in
    match m.mem_opt with
    | `Yes_rem eq when eq v m.mem_descr.default ->
        encode_mems memds mems k e
    | _ ->
        enc_next (`Lexeme (`Name n))
          (m.mem_descr.encode m.mem_descr v (encode_mems memds mems k)) e

let encode_obj objd descr o k e =
  if o.obj_oid <> objd.objd_id then invalid_arg err_oid else
  let pop k e = e.enc_ctx <- List.tl e.enc_ctx; k e in
  e.enc_ctx <- o :: e.enc_ctx;
  enc_next (`Lexeme `Os)
    (encode_mems objd.objd_mem_list o.obj_mems
       (encode_anons objd.objd_anon (Smap.bindings o.obj_anons)
          (enc_next (`Lexeme `Oe) (pop k)))) e

let obj objd =
  objd.objd_used <- true;
  objd.objd_mem_list <- List.rev objd.objd_mem_list; (* dependency order. *)
  let decode descr k d = decode_obj objd descr k d in
  let encode descr k e = encode_obj objd descr k e in
  { default = objd_default objd; decode; encode; }

(* JSON object values *)

let check_mem_oid o m =
  if o.obj_oid <> m.mem_oid then invalid_arg (err_mem_oid m.mem_name) else ()

let get_def m o =
  check_mem_oid o m; m.mem_of_univ (Mmap.find m.mem_id o.obj_mems)
```

```
let get m o = snd (get_def m o)
let set m o v =
  check_mem_oid o m;
  let obj_mems = Mmap.add m.mem_id (m.mem_to_univ (invalid_def v)) o.obj_mems in
  { o with obj_mems }

let check_anon_oid o a =
  if o.obj_oid <> a.anon_oid then invalid_arg err_anon_oid else ()

let anon_names a o =
  check_anon_oid o a; Smap.fold (fun k _ acc -> k :: acc) o.obj_anons []

let find_anon_def a name o =
  check_anon_oid o a;
  try Some (a.anon_of_univ (Smap.find name o.obj_anons)) with
  | Not_found -> None

let find_anon a name o =
  check_anon_oid o a;
  try Some (snd (a.anon_of_univ (Smap.find name o.obj_anons))) with
  | Not_found -> None

let get_anon a name o = match find_anon a name o with
| None -> invalid_arg (err_anon_mem name) | Some v -> v

let get_anon_def a name o = match find_anon_def a name o with
| None -> invalid_arg (err_anon_mem name) | Some v -> v

let add_anon a name o v =
  check_anon_oid o a;
  let obj_anons = Smap.add name (a.anon_to_univ (invalid_def v)) o.obj_anons in
  { o with obj_anons }

let rem_anon a name o =
  check_anon_oid o a;
  let obj_anons = Smap.remove name o.obj_anons in
  { o with obj_anons }

(* JSON object value creation *)

type memv =
  | M : 'a mem * 'a -> memv
  | A : 'a anon * string * 'a -> memv

let memv m v = M (m, v)
let anonv a n v = A (a, n, v)

let new_obj d mems =
  let o = d.default in
  let obj_mems, obj_anons =
    let add (mems, anons) = function
    | M (m, v) ->
        check_mem_oid o m;
        (Mmap.add m.mem_id (m.mem_to_univ (invalid_def v)) mems, anons)
    | A (a, n, v) ->
        check_anon_oid o a;
        (mems, Smap.add n (a.anon_to_univ (invalid_def v)) anons)
    in
    List.fold_left add (o.obj_mems, o.obj_anons) mems
  in
  { o with obj_mems; obj_anons }
```

```
(*---------------------------------------------------------------------------
   Copyright (c) 2014 Daniel C. BÃijnzli.
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions
   are met:

   1. Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.

   2. Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimer in the documentation and/or other materials provided
      with the distribution.

   3. Neither the name of Daniel C. BÃijnzli nor the names of
      contributors may be used to endorse or promote products derived
      from this software without specific prior written permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
   "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
   LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
   A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
   OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
   SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
   LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
   DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
   (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
   OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
  ----------------------------------------------------------------------------*)
```