Thank you!

Examples

Dependent OCaml?

Typed PPX?

Coq?

“Easy” GADT’s

GADT = Existentials + Equality

Existentials

```
type show =
  | Ex_show : {
    content : 'a;
    show : 'a → string
  } → show
```

```
type show =
  | Ex_show : {
    content : 'a;
    show : 'a → string
  } → show
```

show

```
let value = Ex_show {
  content = 1;
  show = Int.to_string
}
```

show -> string

```
let eval_show ex =
  let (Ex_show { content; show }) = ex in
  show content
```

```
show -> string
let eval_show ex =
  let (Ex_show { content; show }) = ex in
  show content
```

$Ex_show_'a

```
show -> string
let eval_show ex =
  let (Ex_show { content; show }) = ex in
  show content
```

$Ex_show_'a → string

```
show -> 'a
let extract ex =
  let (Ex_show { content; show = _ }) = ex in
  content
```

```
This expression has type $Ex_show_'a but an expression was expected of
type
  'a
The type constructor $Ex_show_'a would escape its scope ocamllsp
```

```
module type Show = sig
  type t

  t
  val content : t
  t -> string
  val show : t → string
end

(module Show) -> string
let eval_show package =
  S.t → string                    Show) = package in
  S.show S.content
```

Equalities

```
type (_, _) eq =
  | Refl : ('x, 'x) eq
```

```
type (_, _) eq =
  | Refl : ('x, 'x) eq


let a : (string, string) eq = Refl
let b : (string, int) eq = Refl
```

```
type (_, _) eq =
  | Refl : ('x, 'x) eq

(* weak, how to derive sym? *)
'a 'b. ('a, 'b) eq -> 'a -> 'b
let cast : type a b. (a, b) eq → a → b =
 fun eq x →
  let Refl = eq in
  x
```

```
(* this is how I got here *)
module Subst (M : sig
  type 'a t
end) =
struct
  'a 'b. ('a, 'b) eq -> 'a M.t -> 'b M.t
  let subst : type a b. (a, b) eq → a M.t → b M.t =
   fun eq x →
    let Refl = eq in
    x
end
```

```
(* refutation *)
```

'a. (string, int) eq -> 'a

```
let bad : type a. (string, int) eq → a
  = fun eq → match eq with _ → .
```

```
(* fancier *)
'a. (('a, int) eq, ('a, bool) eq) Either.t -> 'a -> string
let dependent_types :
    type a. ((a, int) eq, (a, bool) eq) Either.t → a → string =
 fun tag content →
  match tag with
  | Left Refl → Int.to_string content
  | Right Refl → Bool.to_string content
```

Just equations

```
type _ ty =
  | T_int : int ty
  | T_bool : bool ty
  | T_list : 'a ty → 'a list ty
```

```
'a. 'a ty -> 'a -> string
let rec show : type a. a ty → a → string =
 fun tag content →
  match tag with
  | T_int → Int.to_string content
  | T_bool → Bool.to_string content
  | T_list el_tag →
      let serialized_content =
        List.map (fun el → show el_tag el) content in
      Format.sprintf "[%s]" (String.concat ", " serialized_content)

let () = print_endline @@ show T_int 1
let () = print_endline @@ show (T_list T_bool) [ true; false ]
```

The Dream

```
type ty = T_int | T_bool | T_list of ty

(* sadly this doesn't work *)
type 'tag to_type =
  [%d
    match tag with
    | T_int → int
    | T_bool → bool
    | T_list el → list (to_type el)]
```

```
ty -> 'a to_type -> string
let rec show tag (content : tag to_type) =
  match tag with
  | T_int → Int.to_string content
  | T_bool → Bool.to_string content
  | T_list el_tag →
      let serialized_content =
        List.map (fun el → show el_tag el) content in
      Format.sprintf "[%s]" (String.concat ", " serialized_content)
```

```
let rec show tag
    (content :
      [%d
        match tag with
        | T_int → int
        | T_bool → bool
        | T_list el → list (to_type el)]) =
```

“Easy” GADT’s

By repeating yourself

```
type t_int = |
type t_bool = |
type _ t_list = |

type _ ty =
  | T_int : t_int ty
  | T_bool : t_bool ty
  | T_list : 'el ty → 'el t_list ty
```

```
type 'tag to_type =
  [%d
    match tag with
    | T_int → int
    | T_bool → bool
    | T_list el → list (to_type el)]

type (_, _) to_type_w =
  | W_t_int : (t_int, int) to_type_w
  | W_t_bool : (t_bool, bool) to_type_w
  | W_t_list : ('tag_el, 'r) to_type_w →
    ('tag_el t_list, 'r list) to_type_w
```

```
'tag 'content. ('tag, 'content) to_type_w -> 'tag ty -> 'content -> string
let rec show :
    type tag content. (tag, content) to_type_w ->
        tag ty -> content -> string
    =
 fun content_type tag content ->
  match tag with
  | T_int ->
      let W_t_int = content_type in
      Int.to_string content
  | T_bool ->
      let W_t_bool = content_type in
      Bool.to_string content
  | T_list el_tag ->
      let (W_t_list el_type) = content_type in
      let serialized_content =
        List.map (fun el -> show el_type el_tag el) content
      in
      Format.sprintf "[%s]" (String.concat ", " serialized_content)
```

```
let rec show :
    type tag content. (tag, content) to_type_w →
        tag ty → content → string
    =
```

```
let rec show :
    type tag. type content = to_type tag.
        tag ty → content → string
    =
```

```
'tag 'content. ('tag, 'content) to_type_w -> 'tag ty -> 'content -> string
let rec show :
    type tag content. (tag, content) to_type_w →
        tag ty → content → string
    =
 fun content_type tag content →
  match tag with
  | T_int →
      let W_t_int = content_type in
      Int.to_string content
  | T_bool →
      let W_t_bool = content_type in
      Bool.to_string content
  | T_list el_tag →
      let (W_t_list el_type) = content_type in
      let serialized_content =
        List.map (fun el → show el_type el_tag el) content
      in
      Format.sprintf "[%s]" (String.concat ", " serialized_content)
```

```
let rec show :
    type tag content. (tag, content) to_type_w →
        tag ty → content → string
    =
```

```
type (_, _) to_type_w =
  | W_t_int : (t_int, int) to_type_w
  | W_t_bool : (t_bool, bool) to_type_w
  | W_t_list : ('tag_el, 'r) to_type_w →
      ('tag_el t_list, 'r list) to_type_w

(* add the original as a package *)
type _ to_type = Ex : ('tag, 'r) to_type_w → 'tag to_type
```

'tag. 'tag ty -> 'tag to_type

```
let rec to_type : type tag. tag ty → tag to_type =
 fun tag →
  match tag with
  | T_int → Ex W_t_int
  | T_bool → Ex W_t_bool
  | T_list el →
      let (Ex el_w) = to_type el in
      Ex (W_t_list el_w)
```

```
let () =
  let tag = T_int in
  let (Ex (type content) (w : (_, content) to_type_w)) = to_type tag in
  let content : content =
    let W_t_int = w in
    1
  in
  print_endline @@ show w tag content

let () =
  let tag = T_list T_bool in
  let (Ex (type content) (w : (_, content) to_type_w)) = to_type tag in
  let content : content =
    let (W_t_list W_t_bool) = w in
    [ true; false ]
  in
  print_endline @@ show w tag content
```

```
let rec show tag (content : tag to_type) =
  match tag with
  | T_int → Int.to_string content
  | T_bool → Bool.to_string content
  | T_list el_tag →
      let serialized_content =
        List.map (fun el → show el_tag el) content in
      Format.sprintf "[%s]" (String.concat ", " serialized_content)

let () = print_endline @@ show T_int 1
let () = print_endline @@ show (T_list T_bool) [ true; false ]
```

Sure … but why?

```
type z = |
type _ s = |
type _ nat = Z : z nat | S : 'a nat → 'a s nat
type (_, _) eq = Refl : ('x, 'x) eq

type ('n, 'm, 'r) add_w =
  | W_n_z : (z, 'm, 'm) add_w
  | W_n_s : ('n, 'm, 'r) add_w → ('n s, 'm, 'r s) add_w

type ('n, 'm) add = Ex : 'r nat * ('n, 'm, 'r) add_w → ('n, 'm) add
```

```
'n 'm. 'n nat -> 'm nat -> ('n, 'm) add
let rec add : type n m. n nat → m nat → (n, m) add =
 fun n m →
  match n with
  | Z → Ex (m, W_n_z)
  | S n →
      let (Ex (r, w)) = add n m in
      Ex (S r, W_n_s w)
```

… sure … but why?