# Ocsigen...

Pierre Chambart

Gallium — March 12th, 2012

- Overview
- Whishlist

# Ocsigen

Basicaly, Ocsigen is an OCaml web framework with a bunch of projects:

- Ocsigen server
- Eliom
- Js_of_ocaml
- Lwt
- Macaque
- Ocismore
- ...

# Disclaimer

In OCaml mainly for historical reasons: Vincent Balat liked it.
It was supposed to be replaced by a specialised, language
But it is quite well suited for what we do.

# General idea

I don't know what are the original ideas behind Ocsigen, but I see what it is now.
Mainly: "you can't be too much wrong when it compiles."

# General idea

I don't know what are the original ideas behind Ocsigen, but I see what it is now.
Mainly: "you can't be too much wrong when it compiles."

(Of course it is always possible to be deliberately wrong)

# Internet is in beta

Web application tend to be quickly written by graphical designers using scripting languages.

This usualy does not help with security.

# Internet is in beta

Web application tend to be quickly written by graphical designers using scripting languages.

This usualy does not help with security.

We would like that kind of users to be able to use Ocsigen.

# Internet is in beta

Web application tend to be quickly written by graphical designers using scripting languages.

This usualy does not help with security.

We would like that kind of users to be able to use Ocsigen.

Yet we are waiting for GADT...

# Ocsigen has a lot of features

We will see only a few.

- Services
- Sessions
- Javascript compilation.

# Safety principle

Type what we can, check at launch time what we can't.

- HTML
- Service parameters
- Service usage
- Database access
- Lots of phantom types

# Droped OCamlduce

Didn't provide much, but added a lot of complexity.

Web services are about generating, not processing XML.

# HTML with polymorphic variants

```
type 'a elt
val pcdata : string -> [> `Pcdata ] elt
val span : [< `Span | `Pcdata ] elt list
  -> [> `Span ] elt
val div : [< `Div | `Span | `Pcdata ] elt list
  -> [> `Div ] elt
```

Can't easilly express all constraints.

# Services

Basicaly a typed URL.

```
let service = register_service
  ~path:["root";"subpage"]
  ~get_params:(int "p1" ** float "p2")
  (fun (p1,p2) () -> ... )

a ~service [pcdata "link"] (1,3.14)
```

# Continuation style

```
let service = register_service ...
  (fun () () ->
    let coservice = register_coservice'
      ~get_params:unit
      (fun () () -> ... ) in
    ...
    a ~service:coservice ()
    ...)
```

# Continuation style

```
let service = register_service ...
  (fun () () ->
    let coservice = register_coservice'
      ~get_params:unit
      (fun () () -> ... ) in
    ...
    a ~service:coservice ()
    ...)
```

Not always a good idea.

# Sessions/scopes

Avoids mixing/leaking the data of users.

```
let site_ref = eref ~scope:site None
let g_ref = eref ~persistent:true ~scope:group 1
let s_ref = eref ~secure:true ~scope:session None
let cp_ref = eref ~scope:client_process 1
```

# Js_of_ocaml

Compile to Javascript.
Uses phantom object types for bindings.

```
class type document = object
  method alert : Js.js_string Js.t -> unit Js.meth
end

let alert s = Dom_html.document##alert(Js.string s)

class type div = object
  method height : int Js.prop
end

div##height <- div##height + 1
```

# Lwt on client side

```
lwt node_content = call_service ~service () () in
replaceAllChilds node node_content;
return ()
```

No need for scheduler.

# One file Client/server app

Heavy camlp4 usage.

```
Server code
{{
    client code
}}
{shared{
    compiled on both
}}
```

# Server side variables access

```
{shared{
  type a = A of int
}}
let x = A 13
let elt =
  div ~onclick:{{let A v = %x in
                 alert(sprintf "alert:%i" v)}}
  [pcdata "div content"]
```

Sends the whole value.
Problem with secret values.

# Client/server typing

Server to client:
Type name is checked.

Client to server:
We use Deriving

# Client/server typing

Server to client:
Type name is checked.

Client to server:
We use Deriving

Different representation of types
Ex: services, HTML, channels, ...
Uses horrible (unsafe) tricks to modify it while marshaling.

Overview

Whishlist

# Whiwhlist

- Marshal closures with dynlink
- Functor pack
- Type error message specialisation
- Runtime types (and more)

# Dynlink everything

```
context := Some (init_context ());
load_extension();
context := None;
```

Problems:

- Some initialisation problems
- Marshalling closures from dynlinked modules (we have a patch)
- Not always natdynlink (solved ?)

The functor pack could help us.

# Custom type errors

```
let x = span [div []];;
                ^^^^^^

Error: This expression has type
  ([> HTML5_types.div ] as 'a) HTML5.M.elt
  but an expression was expected of type
    ([< HTML5_types.span_content_fun ] as 'b) HTML5
  Type 'a = [> `Div ] is not compatible with type
    'b =
      [< `A of HTML5_types.phrasing_without_interac
       ...
       | `Wbr ]
The second variant type does not allow tag(s) `Div
```

I don't have a good proposition for that.

# Runtime types

Extension of Alain Frisch proposition.
Nothing new possible, but makes certain things reasonable.

- Safe unmarshalling. (simpler function definitions)
- Safe type traversal (wrapping).
- Database updates.
- Real value for phatom types.
- Testing usage ?
- ...

# Basics

- New predefined type `'a ty`
  Representation of the type `'a`
- New contructions:
  - `(val of type ...)`
  - `let type val ...  [in]`
- New modules:
  - CamlinternalTy
  - Dynamic

I know, the syntax is awfull.

# val of type

```
let v1 = (val of type int ty)
let v2 = ((val of type _): int ty)

let v3 = ((val of type _): ([`A of 'a] as 'a) ty)

let _ = CamlinternalTy.repr (val of type int ty);;
- : CamlinternalTy.uty =
{expr_id = 20;
 desc =
  DT_constr
   ({decl_id = 3; params = [||]; body = DT_builtin;
     name = "int";
     loc = ("__camlinternalTy__builtin", 0, 0)},
    [||])}
```

# Deconstructing type

```
type _ head =
  | Unit: unit head
  | Int: int head
  | Array: 'a ty -> 'a array head
  | Arrow: 'a ty * 'b ty -> ('a -> 'b) head
  (* ... other predefined types ... *)
  | Tuple: ('a, 'builder) tuple -> 'a head
  | Record: ('a, 'builder) record -> 'a head
  | Variant: 'a variant -> 'a head
  | Opaque: 'a ty option -> 'a head
  | OpaqueCoerce: 'b ty * ('a -> 'b) * ('b -> 'a)
    -> 'a ty

val head: 'a ty -> 'a head
```

# Simple usage

```
let f : type t. t ty -> Format.formatter
  -> t -> unit =
  fun t ppf v ->
  match head t with
   | Unit -> Format.fprintf ppf "()"
   | Int -> Format.fprintf ppf "%d" v
   | Nativeint -> Format.fprintf ppf "%nd" v
   | Int32 -> Format.fprintf ppf "%ld" v
   | Int64 -> Format.fprintf ppf "%Ld" v
   | Char -> Format.fprintf ppf "%C" v
   ...
```

# Deconstruct/build tuples

```
type ('a, 'builder) tuple = private {
  tuple_fields: ('a,'builder) field array;
  tuple_builder: unit -> 'builder;
  tuple_inj: 'builder -> 'a;
}

type (_,_) field = Field:
  ('a,'builder,'b) field_desc ->
  ('a,'builder) field

type ('a,'builder,'b) field_desc = private {
  field_ty: 'b ty;
  field_get: 'a -> 'b;
  field_set: 'builder -> 'b -> unit;
}
```

# let type val

```
let f (ty:'a ty) (t:'a) =
  let type val ty in
  print (val of type 'a list ty) [ t ]
```

# Type association table

We have a table for searching values referenced by types.
Not efficient: We would need a canonical representation of polymorphic variants.

# Type association table

We have a table for searching values referenced by types.
Not efficient: We would need a canonical representation of polymorphic variants.

And we have a better solution.

# Type derivers

We have a generic method for constructing values from type. (in the compiler)

- `(val of type` $\alpha$ `t)` has type $\alpha$ `t`
- `t` must be a type constructor
- in `let type val e`, `e` has type
  - $\alpha$ `t`
  - $\alpha_1 t_1 -> ... -> \alpha_n t_n -> \alpha t$

`let type val` extend the environment
`val of type` search in the environment

# Printer example

```
module Print : sig
  type 'a t
  val print : 'a t -> 'a -> string
  val tot : ('a -> string) -> 'a t
end = struct
  type 'a t = 'a -> string
  let print t a = t a
  let tot x = x end

let type val Print.tot string_of_int
let type val fun p -> Print.tot
  (fun l -> (String.concat "␣" (List.map p l)))
let s = Print.print
  (val of type int Print.t) 1
let s = Print.print
  (val of type int list Print.t) [3; 4]
```

# Multi printer

```
type _ p =
  | Ret : string p
  | Acc : 'a Print.t * 'b p -> ('a -> 'b) p

let rec printer : type t. string -> t p -> t =
  fun acc s -> match s with
    | Ret -> acc
    | Acc (p, r) ->
      fun x -> printer (acc^(Print.print p x)) r
let printer p = printer "" p

let type val fun x r -> Acc (x,r)
let type val Ret
let s1 = printer (val of type string p)
let s2 = printer (val of type (int -> string -> str
  1 "foo"
```

# Implicit argument

```
let display ?#(tx : 'a ty) (x: 'a) = ...
let () = display 3
```

Equivalents to:

```
let () = display ~tx:(type of _) 3
```

# Possible derivers

$\alpha$ `ty` with restrictions

- types that can't contain closure (for marshaling, comparison, ...)
- only tuples of types providing some function (maybe a bit more syntax is needed)
- ...

# Deserialisation

- to/from the client
- database

Live modification of values.

# For HTML

Lightweight syntax for HTML:

```
let elt = toelt (val of type _)
  ("some text", 13, ( "another part", 3.14 ) )

let elt =
  span [pcdata "some text"; pcdata (string_of_int 1
    span [ pcdata "another part"; pcdata (string_of
```

# For HTML

Lightweight syntax for HTML:

```
let elt = toelt (val of type _)
  ("some text", 13, ( "another part", 3.14 ) )

let elt =
  span [pcdata "some text"; pcdata (string_of_int 1
    span [ pcdata "another part"; pcdata (string_of
```

Use different function depending on the context for a more precise HTML type.
Ex: link inside a link, or inside a video

# For HTML

Lightweight syntax for HTML:

```
let elt = toelt (val of type _)
  ("some text", 13, ( "another part", 3.14 ) )

let elt =
  span [pcdata "some text"; pcdata (string_of_int 1
    span [ pcdata "another part"; pcdata (string_of
```

Use different function depending on the context for a more precise HTML type.
Ex: link inside a link, or inside a video

Interface generated from type.

# Lighter service/functions declaration

Types need not to be annotated.

```
let service = register_service
  ~path:["root";"subpage"]
  (fun (p1,p2) () -> ... )
```

Value of phantom type: avoids copying parameter of the registration to the declaration

# Automatic conversion to/from js

Javascript value are different from ocaml ones.
We need to convert values to call Javascript methods.

# Multi type pattern matching ?

Just a possibility ( with some more camlp4 ).

```
type r1 = { r1 : int }
type r2 = { r2 : int }
let f ty v =
match type ty,v with
   | { r1 } -> r1
   | { r2 } -> r2
```

# Almost catch unregistered services

Detect `'_a t` type.

Not a serious protection: it is always possible to force the type.

Can somtimes catch mistakes.

# Both constructions ?

- Derivers useful for handling a particular type. (e.g. plus)
- type representation for generic functions. (e.g. print)

# About the patch

Relatively small patch.
mainly adds.

- add 2 new constructs.
- adds in lambda compilation.
- a few more modifications for implicit argument.

available on gitorious