

Обзор программирования на OCaml для Web

Дмитрий Косарев

JetBrains Labs

It Global Meetup
FProg

17 марта, 2018

План

- Достойное упоминания
- Ocsigen веб-сервер и как заменить Javascript
- Ещё один способ как можно заменить Javascript

Достойная упоминания

Webmachine – для создания RESTful сервисов [▶ Link](#)

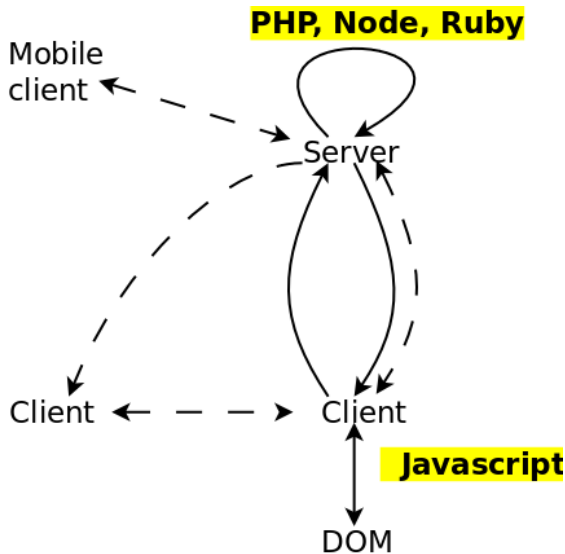
Очень похоже на

- Webmachine для Erlang [▶ Link](#)
- Liberator для Clojure [▶ Link](#)

Что нужно от очередного нового веб-фреймворка на языке \mathcal{L} ?

- Делало то, что нужно
- Своя “фишка”
- Интеграция с кодом на \mathcal{L}
- Интеграция с кодом, который сейчас используется в Web
- Желательно, лаконичность
- Некоторые уважают типобезопасность...
- “Адекватный” синтаксис

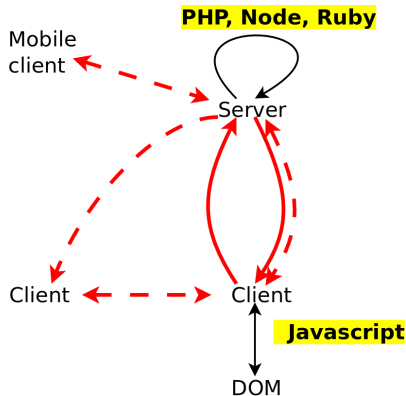
Взаимодействие в Web



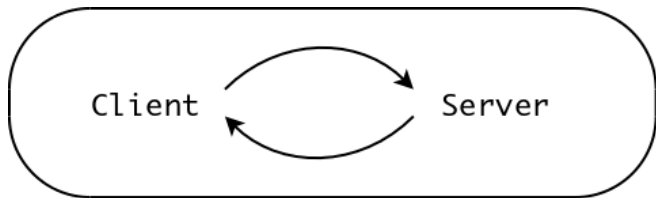
Бестиповость

С сервера послали
1,0:Текст приветствия

На клиенте ожидалось
line <int>: <string>

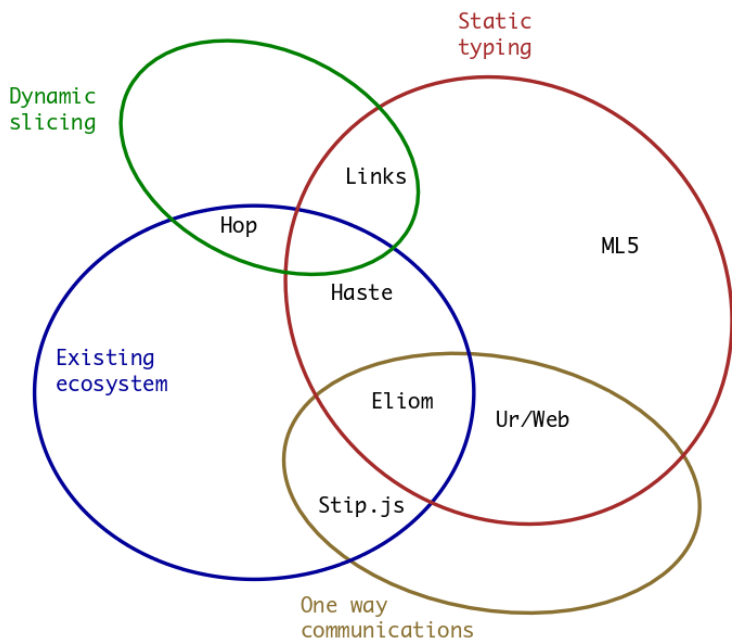


Хочется один tierless язык для всего



Другие tierless языки:

- Links
- Hop
- Ur/Web
- **Eliom**



Срисовано у [Gabriel Radanne](#)



ocrigen
fresh air in web programming

Inria

INVENTEURS DU MONDE NUMÉRIQUE

Eliom	
Server	Js _ of _ ocaml
OCaml	



Library	
Syntax extension	
Server	Js _ of _ ocaml
OCaml	

Серверная часть. Пример 1/2

```
let service_with_params =  
  Eliom_registration.Html.create  
    ~path: ...  
    ~meth:(Get ... )  
    handler
```

Серверная часть. Пример 1/2

```
let service_with_params =  
  Eliom_registration.Html.create  
    ~path: ...  
    ~meth:(Get ... )  
    handler
```

- Именованные параметры
- Можно больше одного сервиса на путь
- В данном случае из handler должен вернуться Html

Серверная часть. Пример 2/2

```
let service_with_params =  
  Eliom_registration.Html.create  
    ~path:(Path ["horde"; "orc"])  
    ~meth:(Get (int "i" ** (int "ii" ** string "s")))  
    (fun (i,(ii,s)) ->  
      (* Тут обрабатываем Get запрос,  
        отдаём Html *)  
    )
```

Серверная часть. Пример 2/2

```
let service_with_params =  
  Eliom_registration.Html.create  
    ~path:(Path ["horde"; "orc"])  
    ~meth:(Get (int "i" ** (int "ii" ** string "s")))  
    (fun (i,(ii,s)) ->  
      (* Тут обрабатываем Get запрос,  
        отдаём Html *)  
    )
```

- Если параметр не подходит по типу – ошибка
- Разумеется, пользовательские типы параметров поддерживаны
- Нетипизированные параметры (список пар строк)

Порождение, например, Html

```
Eliom_registration.Html.create ~path ~meth  
  (fun _ ->  
    let open Eliom_content.Html.D in  
    let input = input ~a:[a_input_type Text] () in
```

Порождение, например, Html

```
Eliom_registration.Html.create ~path ~meth
(fun _ ->
  let open Eliom_content.Html.D in
  let input = input ~a:[a_input_type Text] () in

  let button =
    button ~a:[a_onclick ...] [pcdata "Read value"]
  in
```


Порождение, например, Html

```
Eliom_registration.Html.create ~path ~meth
  (fun _ ->
    let open Eliom_content.Html.D in
    let input = input ~a:[a_input_type Text] () in

    let button =
      button ~a:[a_onclick ...] [pcdata "Read value"]
    in

    Lwt.return
      (html
        (head (title (pcdata "Test")) [])
        (body [input; button])))
  )
```

Типизация через полиморфные варианты

```
(* OK *)  
script ~a:[a_async ()] (pdata "http://something");
```

```
(* ERROR *)  
h2 ~a:[a_async ()] [pdata "Some text"];
```

Больше информации в [OCaml manual](#)

В Haskell их пытаются [эмулировать](#), но выглядит страшно :)

Три семантики для создаваемых тегов

DOM-семантика

Eliom_content.Html.D

“Функциональная”

Eliom_content.Html.F

Реактивная

Eliom_content.Html.R

Если нужно, то можно использовать все три одновременно

D. vs F.

```
let b = button ~a:... [pcdata "text"] in  
div [ b; b; b]
```

По значению vs. по ссылке

Реактивность. Пример

```
let value_signal, set_value = React.S.create "initial"
```

Реактивность. Пример

```
let value_signal, set_value = React.S.create "initial"

let content_signal : _ React.signal =
  React.S.map (fun str ->
    let l = split str in
    let ps = l |> List.map
      (fun s -> F.p [F.pdata s]) in
    F.div ps
  )
value_signal
```

Реактивность. Пример

```
let value_signal, set_value = React.S.create "initial"


let content_signal : _ React.signal =
  React.S.map (fun str ->
    let l = split str in
    let ps = l |> List.map
      (fun s -> F.p [F.pdata s]) in
    F.div ps
  )
  value_signal

let make_client_nodes () =
  [ D.p [R.pdata value_signal]
  ; R.node content_signal
  ]
```

Про self-adjusting вычисления

Incr_dom от [Janestreet Capitals](#) 

Virtual DOM от [Janestreet Capitals](#) 

virtual-dom от [Matt-Esch](#) 

[Блогпост](#) из Janestreet Tech Blog

Фрагменты клиентского и серверного кода

```
let%server x = ...
```

```
let%client y = ...
```

```
let%shared z = ...
```

Всё в одном файле

Фрагменты клиентского кода на сервере

```
let%server x = [%client 1 + 3 ]
```

На клиенте считается 1+3.

Сервер оперирует этим как "черным" ящиком

Доступ к данным сервера на клиенте

```
let%server s : int = 1 + 2
```

```
let%client y : int = ~%s + 1
```

Ограничение – jsonификация типа переменной s

Чуть более содержательный пример

```
let%server counter action =
```

Чуть более содержательный пример

```
let%server counter action =  
  let state = [%client ref 0 ] in
```

Чуть более содержательный пример

```
let%server counter action =  
  let state = [%client ref 0 ] in  
  button ~button_type:Button  
    ~a:[a_onclick [%client fun _ ->
```

Чуть более содержательный пример

```
let%server counter action =  
  let state = [%client ref 0 ] in  
  button ~button_type:Button  
    ~a:[a_onclick [%client fun _ ->  
      incr ~%state;
```

Чуть более содержательный пример

```
let%server counter action =  
  let state = [%client ref 0 ] in  
  button ~button_type:Button  
    ~a:[a_onclick [%client fun _ ->  
      incr ~%state;  
      ~%action ! (~%state)  
    ]]
```


Чуть более содержательный пример

```
let%server counter action =  
  let state = [%client ref 0 ] in  
  button ~button_type:Button  
    ~a:[a_onclick [%client fun _ ->  
      incr ~%state;  
      ~%action ! (~%state)  
    ]]  
  [pdata "Increment"]
```

RPC

```
let%server log str = (* тут логируем str *)
```

RPC

```
let%server log str = (* тут логируем str *)
```

```
let%client log =  
  ~%(Eliom_client.server_function  
    [%derive.json: string]  
    log)
```

RPC

```
let%server log str = (* тут логируем str *)

let%client log =
  ~%(Eliom_client.server_function
    [%derive.json: string]
    log)

let%client () =
  Eliom_client.onload (fun () ->
    (* N.B. Серверные функции недоступны
       пока страничка не прогрузилась *)
    async (fun () ->
      log "Hello from the client to the server!")))
```

Замечание про порядок вычислений

Куски клиентского кода на сервере не вычисляются сразу.
Они регистрируются для последующего исполнения.
Когда страница прислана, то они начинают выполняться.

Объекты Javascript

Объекты Javascript

```
let options = object%js
  val      x = 3 (* read-only prop *)
  val mutable y = 4 (* read/write prop *)
end
```

TyXML API vs. DOM API

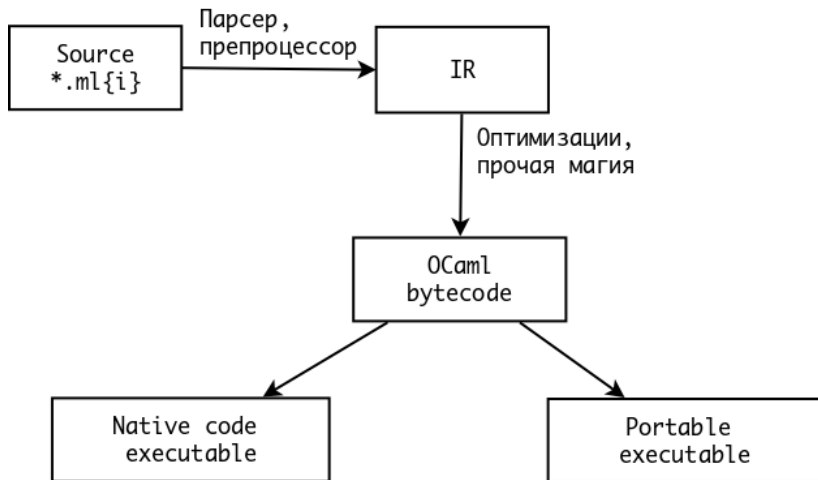
```
let n = div ~a:[a_id "some div id"]  
  [ pcddata "aaa"  
    ; pcddata "bbb" ]  
let d = Eliom_content.Html.To_dom.of_div n
```


TyXML API vs. DOM API

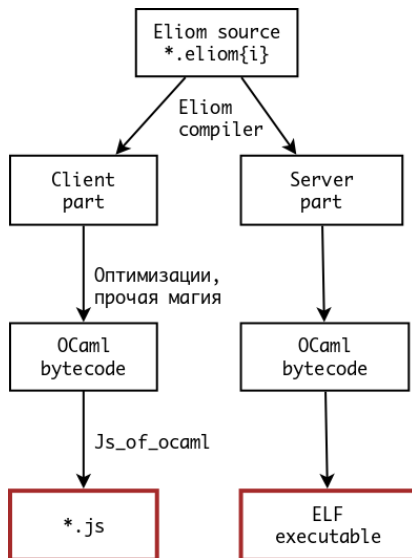
```
let n = div ~a:[a_id "some div id"]  
  [ pcddata "aaa"  
    ; pcddata "bbb" ]  
let d = Eliom_content.Html.To_dom.of_div n
```

```
let d =  
  let d = createDiv document in  
  d##.id := (Js.string "some div id");  
  appendChild d  
    (document##createTextNode (Js.string "aaa"));  
  appendChild d  
    (document##createTextNode (Js.string "bbb"));  
  d
```

Компиляция OCaml кода



Компиляция Eliom проекта



Оптимизации → bytecode → Javascript


Js_of_ocaml генерирует Javascript из **внутреннего** представления (bytecode)

- 👎 Получается нечитаемый код
- 👎 Транслируется не очень быстро

Оптимизации → bytecode → Javascript

Js_of_ocaml генерирует Javascript из **внутреннего** представления (bytecode)

 Получается нечитаемый код

 Транслируется не очень быстро

 Большинство фич языка можно использовать

 Больше оптимизаций, шустрее работает [▶ Графики тут](#)

Следствие: всё что скомпилировалось в байткод, оттранслируется в Javascript

Оптимизации → bytecode → Javascript

Js_of_ocaml генерирует Javascript из **внутреннего** представления (bytecode)

👎 Получается нечитаемый код

👎 Транслируется не очень быстро

👍 Большинство фич языка можно использовать

👍 Больше оптимизаций, шустрее работает [▶ Графики тут](#)

Следствие: всё что скомпилировалось в байткод, оттранслируется в Javascript

Elm 

Ocsigen. Итоги

- “Фишка” – tierless

Ocsigen. Итоги

- “Фишка” – tierless
- Интеграция с существующим кодом на OCaml, **весь** синтаксис поддержан

Ocsigen. Итоги

- “Фишка” – tierless
- Интеграция с существующим кодом на OCaml, **весь** синтаксис поддержан
- Интеграция с существующим кодом на Javascript


Ocsigen. Итоги

- “Фишка” – tierless
- Интеграция с существующим кодом на OCaml, **весь** синтаксис поддержан
- Интеграция с существующим кодом на Javascript
- Типобезопасность в целом

Ocsigen. Итоги

- “Фишка” – tierless
- Интеграция с существующим кодом на OCaml, **весь** синтаксис поддержан
- Интеграция с существующим кодом на Javascript
- Типобезопасность в целом
- Можно оторвать Js_of_ocaml и использовать отдельно

Ocsigen. Итоги

- “Фишка” – tierless
 - Интеграция с существующим кодом на OCaml, **весь** синтаксис поддержан
 - Интеграция с существующим кодом на Javascript
 - Типобезопасность в целом
 - Можно оторвать Js_of_ocaml и использовать отдельно
-  Клиентская часть компилируется небыстро

Ocsigen. Итоги

- “Фишка” – tierless
- Интеграция с существующим кодом на OCaml, **весь** синтаксис поддержан
- Интеграция с существующим кодом на Javascript
- Типобезопасность в целом
- Можно оторвать Js_of_ocaml и использовать отдельно
- 👎 Клиентская часть компилируется небыстро
- 👎 Нечитаемый Javascript на выходе

Ocsigen. Итоги

- “Фишка” – tierless
- Интеграция с существующим кодом на OCaml, **весь** синтаксис поддержан
- Интеграция с существующим кодом на Javascript
- Типобезопасность в целом
- Можно оторвать Js_of_ocaml и использовать отдельно
- 🗨 Клиентская часть компилируется небыстро
- 🗨 Нечитаемый Javascript на выходе
- 🗨 “Адекватный” синтаксис

Ocsigen. Итоги

- “Фишка” – tierless
- Интеграция с существующим кодом на OCaml, **весь** синтаксис поддержан
- Интеграция с существующим кодом на Javascript
- Типобезопасность в целом
- Можно оторвать Js_of_ocaml и использовать отдельно
- 👎 Клиентская часть компилируется небыстро
- 👎 Нечитаемый Javascript на выходе
- 👎 “Адекватный” синтаксис
- 👍 ИМХО, адекватный

Другой вид трансляции

Было:

$*.ml \xRightarrow{OCaml} \text{bytecode} \xRightarrow{Js_of_ocaml} *.js$

Другой вид трансляции

Было:

$*.ml \xRightarrow{OCaml} \text{bytecode} \xRightarrow{Js_of_ocaml} *.js$

А давайте теперь делать вот так: $*.ml \Rightarrow *.js$

- Хотим уметь транслировать **весь** язык

Другой вид трансляции

Было:

$*.ml \xRightarrow{OCaml} \text{bytecode} \xRightarrow{Js_of_ocaml} *.js$

А давайте теперь делать вот так: $*.ml \Rightarrow *.js$

- Хотим уметь транслировать **весь** язык
- 👎 Все OCaml библиотеки надо перекомпилировать
- 👎 Не были использованы OCaml-специфичные оптимизации
- 👍 Есть возможность сделать читаемый Javascript
- 👍 Быстрая (инкрементальная) трансляция

Другой вид трансляции

Было:

$*.ml \xRightarrow{OCaml} \text{bytecode} \xRightarrow{Js_of_ocaml} *.js$

А давайте теперь делать вот так: $*.ml \Rightarrow *.js$

- Хотим уметь транслировать **весь** язык
- 👎 Все OCaml библиотеки надо перекомпилировать
- 👎 Не были использованы OCaml-специфичные оптимизации
- 👍 Есть возможность сделать читаемый Javascript
- 👍 Быстрая (инкрементальная) трансляция

Bucklescript от [Bloomberg](#)

Bucklescript песочница

Bucklescript песочница

По сути backend к компилятору

Bucklescript песочница

По сути backend к компилятору

Иногда, работает быстрее рукописного Javascript


Bucklescript песочница

По сути backend к компилятору

Иногда, работает быстрее рукописного Javascript

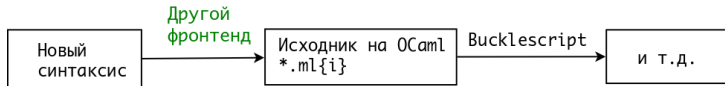
Js_of_ocaml дружелюбнее к инфраструктуре OCaml,
BuckleScript – нет

Почти все проблемы решены...

 Кроме массового недовольства синтаксисом

Почти все проблемы решены...

🗨️ Кроме массового недовольства синтаксисом



ReasonML

Сайт


Синтаксис Reason vs. OCaml

Песочница

ReasonConf 11–13 мая 2018, Вена, Австрия

Вопросы?

Какие-то ссылки...

Ocsigen demo  и [online](#)

Ocsigen [graffiti](#) demo 