Formlet : un des pilliers de la programmation web

ou comment contrôler la description et la validation de formulaires

Xavier Van de Woestyne

Front(end | &) Beers

6 Octobre 2020

Xavier Van de Woestyne

Informations

- Bruxelles, Paris, Nantes
- `https://xvw.github.io`
- @vdwxv, xvw@merveilles.town

Technologie

- *Data Engineer* (mais je préfère le *front*)
- J'aime bien la programmation (idéalement) fonctionnelle statiquement typée
- **OCaml**, Elixir, **Kotlin**, Elm, Haskell

Memo Bank

Première banque indépendante de tout établissement bancaire est lancée en France depuis 50 ans

`https://memo.bank`

Kotlin, Elixir, React et TypeScript

Questions préliminaires

Qu'est-ce qu'un *Data Engineer* pourrait dire à une conférence sur le développement *frontend* ?

Questions préliminaires

Qu'est-ce qu'un *Data Engineer* pourrait dire à une conférence sur le développement *frontend* ?

Probablement pas grand chose ... (en tant que *Data Engineer*).

Questions préliminaires

Qu'est-ce qu'un *Data Engineer* pourrait dire à une conférence sur le développement *frontend* ?

Probablement pas grand chose ... (en tant que *Data Engineer*).

Quels sont les réels objectifs de la présentation ?

Questions préliminaires

Qu'est-ce qu'un *Data Engineer* pourrait dire à une conférence sur le développement *frontend* ?

Probablement pas grand chose ... (en tant que *Data Engineer*).

Quels sont les réels objectifs de la présentation ?

- Faire la promotion des langages fonctionnels statiquement typés ;

Questions préliminaires

Qu'est-ce qu'un *Data Engineer* pourrait dire à une conférence sur le développement *frontend* ?

Probablement pas grand chose ... (en tant que *Data Engineer*).

Quels sont les réels objectifs de la présentation ?

- Faire la promotion des langages fonctionnels statiquement typés ;
- présenter des abstractions réutilisables ;

Questions préliminaires

Qu'est-ce qu'un *Data Engineer* pourrait dire à une conférence sur le développement *frontend* ?

Probablement pas grand chose ... (en tant que *Data Engineer*).

Quels sont les réels objectifs de la présentation ?

- Faire la promotion des langages fonctionnels statiquement typés ;
- présenter des abstractions réutilisables ;
- les formulaires sont un prétexte.

Pourquoi specifiquement traiter les formulaires

- Un très bon cas d'école
- A l'intersection entre **le métier**, **le développement** et **l'expérience utilisateur**
- Rempli de *Dark Pattern*, il faut donc mettre des outils robustes à disposition pour les éviter
- En vrai, je trouve le sujet rigolo

A quoi bon connaitre des abstractions

Généralement, il existe (sûrement dans le monde JavaScript) une collection de bibliothèques qui exploitent intelligemment ces abstractions. Par exemple, `useForm` pour React/Redux... Donc pourquoi perdre du temps à les apprendre et les comprendre ?

Un développeur pragmatique en 2020

A quoi bon connaitre des abstractions

- *Pour briller en société*

A quoi bon connaitre des abstractions

- *Pour briller en société*
- Dans le cas où notre technologie n'aurait pas de bibliothèque (passer de React à Vue par exemple)

A quoi bon connaitre des abstractions

- *Pour briller en société*
- Dans le cas où notre technologie n'aurait pas de bibliothèque (passer de React à Vue par exemple)
- **Parce que généralement, ces abstractions capturent plus qu'un seul *usecase***

A quoi bon connaitre des abstractions

- *Pour briller en société*
- Dans le cas où notre technologie n'aurait pas de bibliothèque (passer de React à Vue par exemple)
- **Parce que généralement, ces abstractions capturent plus qu'un seul *usecase***
- **Parce qu'après, il est difficile de s'en passer**

Si elles sont si bien pourquoi ne sont-elles pas partout ?

- Elles peuvent être **intimidantes** [1]
- Certains langages rendent leur utilisation complexe parce que absence de :
  - Curryfication
  - système de types algébriques

1 - Le but de cette présentation est d'en démystifier au travers d'exemples concrets !

Une première abstraction : **un Foncteur**

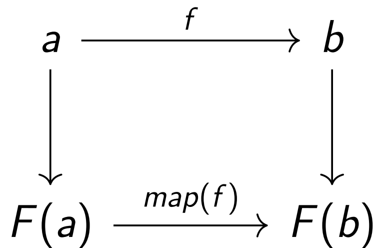
$$
\begin{array}{ccc}
a & \xrightarrow{f} & b \\
\downarrow & & \downarrow \\
F(a) & \xrightarrow{map(f)} & F(b)
\end{array}
$$

Une première abstraction : **un Foncteur**

$$\begin{array}{ccc} a & \xrightarrow{f} & b \\ \downarrow & & \downarrow \\ F(a) & \xrightarrow{map(f)} & F(b) \end{array}$$

Un exemple moins méchant qu'il n'y parait

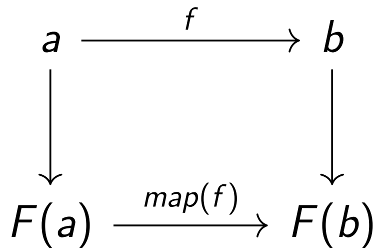
Une première abstraction : **un Foncteur**



Un exemple moins méchant qu'il n'y parait

- `a` et `b` sont des variables de types

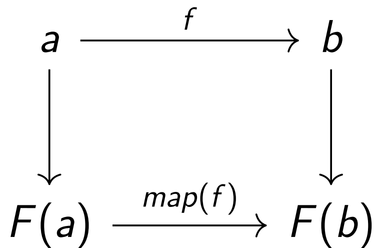
Une première abstraction : **un Foncteur**



Un exemple moins méchant qu'il n'y parait

- `a` et `b` sont des variables de types
- `F(a)` et `F(b)` sont des constructeurs de types

Une première abstraction : **un Foncteur**



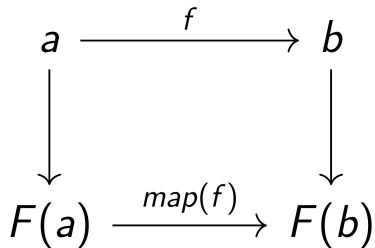
Un exemple moins méchant qu'il n'y parait

- `a` et `b` sont des variables de types
- `F(a)` et `F(b)` sont des constructeurs de types

ce qui se traduit (informellement) par

Si j'ai une fonction qui va de `a` vers `b` et une type `F` paramétré qui a une fonction `map` on peut aller de `F(a)` vers `F(b)`.

Une première abstraction : **un Foncteur**



Un exemple moins méchant qu'il n'y parait

- `a` et `b` sont des variables de types
- `F(a)` et `F(b)` sont des constructeurs de types

ce qui se traduit (informellement) par

Si j'ai une fonction qui va de `a` vers `b` et une type `F` paramétré qui a une fonction `map` on peut aller de `F(a)` vers `F(b)`.

par exemple :

```
[1, 2, 3].map((x) => x.toString())
```

A l'usage

```
const users = [
    {name: 'Xvw', age: 30},
    {name: 'Grim', age: 25},
];

const toUserHTML = (user) => (
    <li>
      {user.name} ({user.age} ans)
    </li>
)

const UserList = () => (
    <div>
        <ul>{users.map(toUserHTML)}</ul>
    </div>
);
```

Soit aller d'une liste d'utilisateurs à une liste de noeuds HTML

Alerte

A partir de maintenant, le code sera en **OCaml** pour plus de lisibilité. **N'hésitez surtout pas à m'interrompre si certains points syntaxiques ne sont pas clairs !**

Concrètement un Foncteur doit respecter cette interface

```
module type FUNCTOR = sig
  (* le type du foncteur, pour liste, `'a t` = `'a list`. *)
  type 'a t

  (* la fonction de mapping. *)
  val map : ('a -> 'b) -> 'a t -> 'b t
end
```

Concrètement un Foncteur doit respecter cette interface

```ocaml
module type FUNCTOR = sig
  (* le type du foncteur, pour liste, `'a t` = `'a list`. *)
  type 'a t

  (* la fonction de mapping. *)
  val map : ('a -> 'b) -> 'a t -> 'b t
end
```

Des lois

En complément, un foncteur doit respecter certaines lois, cependant nous ne nous y attarderons pas dans cette présentation.

Ces lois permettent, entre autre de générer des fonctions usuelles, par exemple :

```ocaml
let replace x f = map (fun _ -> x) f
```

Deux autres foncteurs usuels

Option

```
module Option = struct
  type 'a t =
    | Some of 'a
    | None

  let map f x =
    match x with
    | None -> None
    | Some value -> Some (f value)
end
```

Result

```
module Result = struct
  type 'a t =
    | Success of 'a
    | Error   of exn

  let map f x =
    match x with
    | Error err -> Error err
    | Success value -> Success (f value)
end
```

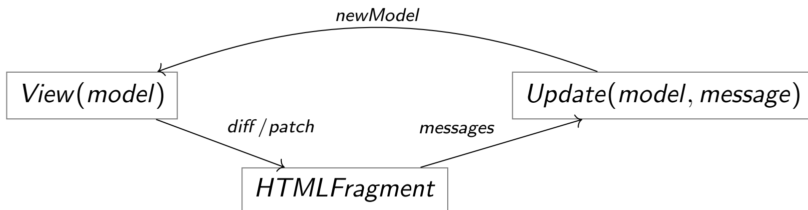
Rentrons dans le vif du sujet : les formulaires

Construire la représentation HTML d'un formulaire est assez simple :

- On projette le **but** du formulaire dans un type
- On fourni une fonction de transformation du **but** vers un fragment HTML

```
module Registration = struct
  type input = {
    username : string;
    password : string;
    email : string;
    age : int;
  }
end
```

Supposons que notre application est structurée par une **machine de moore**



Pour les besoins de la présentation

Au passage, ce genre de machine est assez facile à programmer ... (même si le diff/patch peut être un peu plus tênu)

Rendu du type en formulaire

```
let render_form =
  let open Tyxml.Html in
  form
    [
      input ~a:[ a_input_type `Text; a_placeholder "username" ] ();
      input ~a:[ a_input_type `Password ] ();
      input ~a:[ a_input_type `Email; a_placeholder "email" ] ();
      input ~a:[ a_input_type `Number; a_placeholder "age" ] ();
    ]
```

- On suppose qu'il existe des Handlers implicites (pour ne pas alourdir le code).
- Le submit n'est pas rendu pour pour permettre la composition

Pour les plus fainéants

```
module Registration = struct
  type input = {
    username : string;
    password : string; [@kind "password"]
    email : string; [@kind "email"]
    age : int;
  }
  [@@deriving "formlet"]
end
```

Dérivation automatique de formulaires

Il est assez facile d'imaginer une stratégie de dérivation.

Comment calculer la dérivation

- Traverse récursivement l'AST et collecte les `formlet`
- Enrichi la fonction d'update de la machine pour gérer l'état local des inputs (via, par exemple, un UUID et un Hashtable)

La traversée récursive des formulaires rend leur composition triviale

```
type comp = (form1 * form2) [@@deriving "formlet"]

type comp2 = {
  registration: Registration.input;
  form2: Form2;
  confirm_email: [@kind "email"]
} [@@deriving "formlet"]
```

Une approche plus **idiomatique** de la construction/composition

Il existe d'autre approches pour construire/composer les formulaires, mais elles reposent sur des abstractions que nous verrons plus tard.

Il y a une blague dans cette slide... discrète

Valider nos formulaires

Pourquoi ne pas simplement typer ***sémantiquement*** *et* ***finement*** *nos formulaire avec les filtres HTML ?*

Valider nos formulaires

Pourquoi ne pas simplement typer ***sémantiquement*** *et* ***finement*** *nos formulaire avec les filtres HTML ?*

Très bonne question !

Valider nos formulaires

*Pourquoi ne pas simplement typer **sémantiquement** et **finement** nos formulaire avec les filtres HTML ?*

Très bonne question !

- Il ne faut pas s'en priver, mais, selon moi, ce n'est pas suffisant
- peu de contrôle sur le flot des validation
- limité par le navigateur (et potentiellement pas raccord chez tout le monde)
- Dédouble le code (et oui, pourquoi ne pas utiliser le même code côté *backend* et *frontend* ?)

Rappelons-nous nos deux foncteurs complémentaires

```
module Result = struct
  type 'a t =
    | Success of 'a
    | Error   of exn

  let map f x =
    match x with
    | Error err -> Error err
    | Success value -> Success (f value)
end
```

`Result` nous permettrait de construire des **calculs qui peuvent échouer**

Par exemple

```ocaml
exception Division_by_zero

(* val safe_div : int -> int -> int Result.t  *)
let safe_div numerator denominator =
  if denominator = 0 then Error Division_by_zero
  else Success (numerator / denominator)

exception Invalid_string of string

(* val int_from_string : string -> int Result.t *)
let int_from_string value =
  match int_of_string_opt value with
  | Some x -> Success x
  | None -> Invalid_string value
```

Et à l'usage

```
(* val computation : string Result.t *)
let computation =
  int_from_string "123456"
  |> Result.map successor
  |> Result.map string_from_int
  |> Result.map String.capitalize
```

Et à l'usage

```ocaml
(* val computation : string Result.t *)
let computation =
  int_from_string "123456"
  |> Result.map successor
  |> Result.map string_from_int
  |> Result.map String.capitalize
```

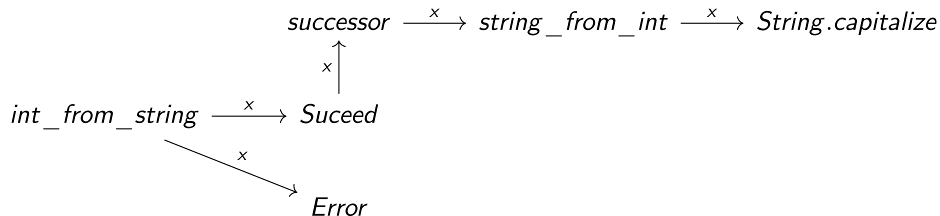
Un petit problème...

Et à l'usage

```
(* val computation : string Result.t *)
let computation =
  int_from_string "123456"
  |> Result.map successor
  |> Result.map string_from_int
  |> Result.map String.capitalize
```

Un petit problème...

- On est bloqué dans un pipeline de succès
- `int_from_string x |> Result.map safe_div 10` renverra un `int Result.t Result.t`



On est bloqué :/

`Result.map` ne peut exprimer que le *Happy Path*. On voudrait pourvoir exprimer un échec à chaque branche.

Introduisons un nouvel outil

```
module Result = struct
- (* val map : ('a -> 'b) -> 'b Result.t *)
+ (* val and_then : ('a -> 'b Result.t) -> 'b Result.t *)
  let and_then f x =
    match x with
    | Error err -> Error err
-   | Success value -> Success (f value)
+   | Success value -> (f value)
end
```

Introduisons un nouvel outil

```
module Result = struct
- (* val map : ('a -> 'b) -> 'b Result.t *)
+ (* val and_then : ('a -> 'b Result.t) -> 'b Result.t *)
  let and_then f x =
    match x with
    | Error err -> Error err
-   | Success value -> Success (f value)
+   | Success value -> (f value)
end
```

La fonction ressemble à `map` sauf que c'est **la fonction passée en argument qui décide de sa réussite ou son échec**.

Cet outil est plus puissant que map

Cet outil est plus puissant que map

```
module Result = struct
  type 'a t =
    | Success of 'a
    | Error   of exn

  (* val and_then : ('a -> 'b Result.t) -> 'b Result.t *)
  let and_then f x =
    match x with
    | Error err -> Error err
    | Success value -> (f value)

  (* val map : ('a -> 'b) -> 'b Result.t *)
  let map f = and_then (fun x -> Success (f x))
end
```

On peut maintenant utiliser des fonctions simples et des fonctions échouables

```
(* val computation : string Result.t *)
let computation =
  int_from_string "123456"
  |> Result.map successor
  |> Result.and_then (safe_div 15)
  |> Result.map string_from_int
  |> Result.map String.capitalize
```

Un peu de sucre syntaxique

Il est commode de définir des version infixes de `map` et `and_then`

```
let ( >>= ) x f = Result.and_then f x
let ( >|= ) x f = Result.map f x
```

Un peu de sucre syntaxique

Il est commode de définir des version infixes de map et and_then

```
let ( >>= ) x f = Result.and_then f x
let ( >|= ) x f = Result.map f x
```

```
(* val computation : string Result.t *)
let computation =
  int_from_string "123456"
  >|= successor
  >>= (safe_div 15)
  >|= Result.map string_from_int
  >|= Result.map String.capitalize
```

Un premier cas d'usage

1 ⌄ Janvier ⌄ 1970 ⌄

- Année doit être comprise entre 1970 et 2020
- La date doit être valide

Implémentons les validateurs

L'année

```
exception Invalid_year of int

let validate_year x =
  int_from_string x
  >>= (fun year ->
    if year >= 1970 && year <= 2020 then Success year
    else Error (Invalid_year year)
  )
```

Implémentons les validateurs

Le mois

```
exception Invalid_month of int

let validate_month year x =
  int_from_string x
  >>= (fun month ->
    if month >= 1 && month <= 12 then Success (year, month)
    else Error (Invalid_month month)
  )
```

Implémentons les validateurs

Le jour

```
exception Not_a_bissextile_year of int
exception Day_negative_or_null of int
exception To_high_for_month of (int * int)

let validate_day (year, month) x =
  int_from_string x
  >>= (fun day ->
    if day < 1 then Error (Day_negative_or_null day)
    else if not is_bissextile year && month = 2 && day > 28 then
      Error (Not_a_bissextile_year year)
    else if day > days_of month then Error (To_high_for_month (month, day))
    else Success (year, month, day)
  )
```

La validation complète

```
(*
   val validate_date :
       string
    -> string
    -> string
    -> (int * int * int) Result.t
*)
let validate_date year month day =
  (validate_year year)
  >>= validate_month
  >>= validate_day day
```

Superbe !

Nous pouvons composer nos filtres sur un triplet ! Validons maintenant qu'un mot de passe fait :

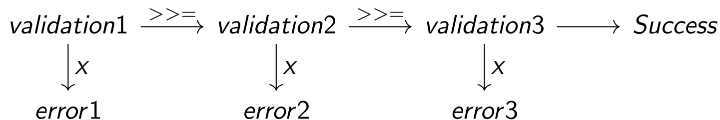
- Au moins 8 caractères
- Contient au moins une majuscule
- Contient au moins un nombre

Superbe !

Nous pouvons composer nos filtres sur un triplet ! Validons maintenant qu'un mot de passe fait :

- Au moins 8 caractères
- Contient au moins une majuscule
- Contient au moins un nombre

Alerte DARK-PATTERN



Une question de flot

Dès qu'une erreur survient, le flot est **court-circuité**. Ce qui était parfait pour la construction de la date, beaucoup moins pour la validation d'un mot de passe.

Entre séquentialité et parallélisme

- Certains composants du formulaire sont **inter-connectés**
- D'autres sont **indépendants**

Entre séquentialité et parallélisme

- Certains composants du formulaire sont **inter-connectés**
- D'autres sont **indépendants**
- Ce qui sous-entend que certains composants sont résolus **séquentiellement** (la date par exemple, où le mois et l'année doivent être calculés pour pour valider le jours)
- D'autres sont résolus **parallèlement** (par exemple plusieurs filtres sur une même valeurs, ou chaque composant d'un formulaire)
- les composants résolus parallèlement doivent **accumuler les erreurs**

Introduisons une nouvelle construction

```
module Result = struct
  type 'a t =
    | Success of 'a
    | Error   of exn

  let and_then f x =
    match x with
    | Error err -> Error err
    | Success value -> (f value)

  let map f =
    and_then (fun x -> Success (f x))
end
```

```
module Validation = struct
  type 'a t =
    | Success of 'a
    | Errors of exn list

  let and_then f x =
    match x with
    | Error err -> Error err
    | Success value -> (f value)

  let map f =
    and_then (fun x -> Success (f x))
end
```

On peut facilement fournir des conversions de l'un à l'autre.

On reçoit une liste au lieu d'un erreur, oké...

Mais comment **accumuler les erreurs** ?

On reçoit une liste au lieu d'un erreur, oké...

Mais comment **accumuler les erreurs** ?

```
module Validation = struct

  let and_then f x =
    match x with
-    | Error err -> Error err
+    | Error err -> ????
     | Success value -> (f value)

end
```

Introduisons un nouvel outil

```
module Validation = struct
+  let apply fx xs =
+     match (fx, xs) with
+     | (Success f, Success x) -> Ok (f x)
+     | (Errors l, Errors r) -> Errors (l @ r)
+     | (Errors x, _) | (_, Errors x) -> Errors x

     (* Pour lequel on a aussi un infix *)
+   let ( <*> ) = apply
end
```

Introduisons un nouvel outil

```
module Validation = struct
+   let apply fx xs =
+      match (fx, xs) with
+      | (Success f, Success x) -> Ok (f x)
+      | (Errors l, Errors r) -> Errors (l @ r)
+      | (Errors x, _) | (_, Errors x) -> Errors x

     (* Pour lequel on a aussi un infix *)
+    let ( <*> ) = apply
end
```

Observons sa signature

```
val apply : ('a -> 'b) t -> 'a t -> 'b t
```

En fait une fonction de `'x -> 'y -> 'z`
est une fonction `'a -> 'b` : où `'x = 'a` et `('y -> 'z) = 'b)`.

Concrètement

- `apply` prend une fonction **emballée** dans une Validation
- une valeur emballée dans une Validation

Concrètement

- `apply` prend une fonction **emballée** dans une Validation
- une valeur emballée dans une Validation
- Si les deux sont valides, elle déballe la fonction et l'applique à la valeur et la remballe dans une Validation (Success)

Concrètement

- `apply` prend une fonction **emballée** dans une Validation
- une valeur emballée dans une Validation
- Si les deux sont valides, elle déballe la fonction et l'applique à la valeur et la remballe dans une Validation (Success)
- Si l'une des deux valeurs est invalides et la renvoie

Concrètement

- `apply` prend une fonction **emballée** dans une Validation
- une valeur emballée dans une Validation
- Si les deux sont valides, elle déballe la fonction et l'applique à la valeur et la remballe dans une Validation (Success)
- Si l'une des deux valeurs est invalides et la renvoie
- Si les deux valeurs sont invalides, elle les concatère et les renvoies emaballés dans une erreur.

Par exemple

```
let create_human age name email = {
    age = age;
    name = name;
    email = email;
  }

let validate_human age name email =
  create_human
  >|= validate_age age
  <*> validate_name name
  <*> validate_email email
```

- Si tout est correcte : `Success {name; age; email}`

Par exemple

```
let create_human age name email = {
    age = age;
    name = name;
    email = email;
  }

let validate_human age name email =
  create_human
  >|= validate_age age
  <*> validate_name name
  <*> validate_email email
```

- Si tout est correcte : `Success {name; age; email}`
- Si l'age et le nom sont incorrectes : `Errors [Invalid_age; Invalid_name]`

On arrive au bout

- apply et and_then permettent de composer arbitrairement des fragments de formulaires adjoints à des validations

On arrive au bout

- apply et and_then permettent de composer arbitrairement des fragments de formulaires adjoints à des validations
- ces validations peuvent s'utiliser avec toute autre forme de choses à valider (par exemple du JSON)

On arrive au bout

- apply et and_then permettent de composer arbitrairement des fragments de formulaires adjoints à des validations
- ces validations peuvent s'utiliser avec toute autre forme de choses à valider (par exemple du JSON)
- and_then capture la séquentialité

On arrive au bout

- apply et and_then permettent de composer arbitrairement des fragments de formulaires adjoints à des validations
- ces validations peuvent s'utiliser avec toute autre forme de choses à valider (par exemple du JSON)
- and_then capture la séquentialité
- apply capture le parallèlisme

Pour conclure

Le foncteur

```
module type FUNCTOR = sig
  (* le type du foncteur, pour liste, `'a t` = `'a list`. *)
  type 'a t

  (* la fonction de mapping. *)
  val map : ('a -> 'b) -> 'a t -> 'b t
end
```

Pour conclure

Quand la structure possède un Apply, c'est un foncteur Applicatif

```
module type APPLICATIVE = sig
  type 'a t

  val apply : ('a -> 'b) t -> 'a t -> 'b t
end
```

Pour conclure

Quand la structure possède un AndThen, c'est une Monade

```
module type MONAD = sig
  type 'a t

  val flat_map : ('a -> 'b t) -> 'a t -> 'b t
end
```

Pour conclure.

Vous devriez vous intéressez à Traverse, après demain à LambdaLille

Références I

Conor McBride and Ross Paterson.
Applicative programming with effects.
http://strictlypositive.org/IdiomLite.pdf.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop.
The Essence of Form Abstraction.
http://homepages.inf.ed.ac.uk/slindley/papers/formlets-essence.pdf.

Jeremy Yallop.
Abstraction for web programming.
https://www.cl.cam.ac.uk/~jdy22/papers/dissertation.pdf.

Jeremy Gibbons and Bruno C. d. S. Oliveira.
The Essence of the Iterator Pattern.
https://www.cs.ox.ac.uk/jeremy.gibbons/publications/iterator.pdf.

Références II

Loïc Denuzière, Adam Granicz, and Simon Fowler.

Reactive Abstractions for Functional Web Applications.

http://www.simonjf.com/drafts/reactive-abstractions.pdf.