Generating Static Websites the Functional Programming Way

Xavier Van de Woestyne | xvw.lol | Tarides

Generating Static Websites the **Functional Programming** Way

Xavier Van de Woestyne | **xvw.lol** | Tarides

Making Critical Systems Better

We help our clients build **reliable**, **secure**, and **high-performance** systems.

(We are strongly involved into the OCaml ecosystem)

Generating Static Websites the **Functional Programming** Way

Xavier Van de Woestyne | **xvw.lol** | Tarides

Making Critical Systems Better

We help our clients build **reliable**, **secure**, and **high-performance** systems.

(We are strongly involved into the OCaml ecosystem)

But this presentation has **nothing to do** with my work.

I mainly work on the **editor support** (**Merlin**, **LSP**, **Emacs**)

Theory and practice behind a **Build-System approach** to static site generation

Generating Static Websites the Functional Programming Way

Xavier Van de Woestyne | **xvw.lol** | Tarides

Making Critical Systems Better

We help our clients build **reliable**, **secure**, and **high-performance** systems.

(We are strongly involved into the OCaml ecosystem)

But this presentation has **nothing to do** with my work.

I mainly work on the **editor support** (**Merlin**, **LSP**, **Emacs**)

Generating Static Websites the **Functional Programming** Way

Xavier Van de Woestyne | **xvw.lol** | Tarides

Theory and practice behind a **Build-System approach** to static site generation

on top of `YOCaml` !

which has recently finally been given a tutorial:
https://yocaml.github.io/tutorial

Making Critical Systems Better

We help our clients build **reliable**, **secure**, and **high-performance** systems.

(We are strongly involved into the OCaml ecosystem)

I mainly work on the **editor support** (**Merlin**, **LSP**, **Emacs**)

But this presentation has **nothing to do** with my work.

Static site generator?

A tool that **processes documents** to **produce an entire website without requiring additional server processing** to serve pages.
(It is therefore a **highly specialised build system**.)

Static site generator?

A tool that **processes documents** to **produce an entire website without requiring additional server processing** to serve pages.
(It is therefore a **highly specialised build system**.)

- super **easy to deploy**
- **restriction** on interaction (JAM)
- **makes trivial things extremely difficult**

Static site generator?

A tool that **processes documents** to **produce an entire website without requiring additional server processing** to serve pages.
(It is therefore a **highly specialised build system**.)

- super **easy to deploy**
- **restriction** on interaction (JAM)
- **makes trivial things extremely difficult**

like **backlinks**
and **transclusion**

`Why use YOCaml?`

Static site generator?

A tool that **processes documents** to **produce an entire website without requiring additional server processing** to serve pages. (It is therefore a **highly specialised build system**.)

- super **easy to deploy**
- **restriction** on interaction (JAM)
- **makes trivial things extremely difficult**

like **backlinks** and **transclusion**

`Why use YOCaml?`

- Please, **stop using Medium**
- It is **highly customizable**
- It is fun (and in **OCaml**)
- It can be a **permanent Project**
- **It is fun** (2)

Static site generator?

A tool that **processes documents** to **produce an entire website without requiring additional server processing** to serve pages. (It is therefore a **highly specialised build system**.)

- super **easy to deploy**
- **restriction** on interaction (JAM)
- **makes trivial things extremely difficult**

like **backlinks** and **transclusion**

`Why use YOCaml?`

- Please, **stop using Medium**
- It is **highly customizable**
- It is fun (and in **OCaml**)
- It can be a **permanent Project**
- **It is fun** (2)

Static site generator?

A tool that **processes documents** to **produce an entire website without requiring additional server processing** to serve pages. (It is therefore a **highly specialised build system**.)

- super **easy to deploy**
- **restriction** on interaction (JAM)
- **makes trivial things extremely difficult**

like **backlinks**
and **transclusion**

The internet is **fun** when you're building personal websites.
Honestly, **The Geocities era is so much better than Medium**!

Create a static blog generator is easy

Create a static blog generator is easy

blog.ml

```
let () =
 let file = Sys.argv.(1) in
 let file_html =
   file
   |> Filename.basename
   |> Filename.remove_extension
 in
 let target = "_www/" ^ file_html in
 let (metadata, content) = File.read file in
 let markdown = Markdown.of_string content in
 let injected =
   Template.inject
   "article.html"
   metadata
   markdown
 in File.write target injected
```

Create a static blog generator is easy

blog.ml

```
let () =
 let file = Sys.argv.(1) in
 let file_html =
   file
   |> Filename.basename
   |> Filename.remove_extension
 in
 let target = "_www/" ^ file_html in
 let (metadata, content) = File.read file in
 let markdown = Markdown.of_string content in
 let injected =
   Template.inject
   "article.html"
   metadata
   markdown
 in File.write target injected
```

Makefile

```
_www/images/%.png: images/%.png
   cp $(<) $(@)

_www/%.css: css/%.png
   cp $(<) $(@)

_www/posts/%.md: _www/%.html
   dune exec blog.exe -- $(<)
```

Create a static blog generator is easy

blog.ml

```
let () =
 let file = Sys.argv.(1) in
 let file_html =
   file
   |> Filename.basename
   |> Filename.remove_extension
 in
 let target = "_www/" ^ file_html in
 let (metadata, content) = File.read file in
 let markdown = Markdown.of_string content in
 let injected =
   Template.inject
   "article.html"
   metadata
   markdown
 in File.write target injected
```

Makefile

```
_www/images/%.png: images/%.png
    cp $(<) $(@)

_www/%.css: css/%.png
    cp $(<) $(@)

_www/posts/%.md: _www/%.html
    dune exec blog.exe -- $(<)
```

That's it
Let's go further

Program

How to be generic

Execution abstraction with **effects and an IO monad**.
Data model abstraction with **applicative validation**.

Program

Program

How to be generic

Execution abstraction with **effects and an IO monad**.
Data model abstraction with **applicative validation**.

How to be minimal

Handling Static dependencies using Arrows
(Strong Profunctor + Category)

Program

How to be generic

Execution abstraction with **effects and an IO monad**.
Data model abstraction with **applicative validation**.

How to be minimal

Handling Static dependencies using Arrows
(Strong Profunctor + Category)

~~How to be extensible~~

~~Reasoning behind the **extensibility**, to integrate features inspired by the Xanadu project.~~

If we have time

Execution abstraction

WHY?

Execution abstraction

- **Trivial reason:** Unix, Windows, abstracting over the platform
- **Drastically facilitates testability** (a test-context is just another platform)
- Supports **exotic target types** (ie: GIT + Mirage)

WHY?

Execution abstraction

HOW?

- **Trivial reason:** Unix, Windows, abstracting over the platform
- **Drastically facilitates testability** (a test-context is just another platform)
- Supports **exotic target types** (ie: GIT + Mirage)

WHY?

Execution abstraction

- **Manual encoding:** Functors/FCM, OOP, Functions
- **Effect System**: Free/Freer, Tagless Final, ReaderT
- **OCaml User Defined Effects**

HOW?

- **Trivial reason:** Unix, Windows, abstracting over the platform
- **Drastically facilitates testability** (a test-context is just another platform)
- Supports **exotic target types** (ie: GIT + Mirage)

WHY?

Execution abstraction

Execution abstraction

WHY?

- **Trivial reason:** Unix, Windows, abstracting over the platform
- **Drastically facilitates testability** (a test-context is just another platform)
- Supports **exotic target types** (ie: GIT + Mirage)

HOW?

- **Manual encoding:** Functors/FCM, OOP, Functions
- **Effect System:** Free/Freer, Tagless Final, ReaderT
- **OCaml User Defined Effects**

We picked this for the hype

- **Manual encoding:** Functors/FCM, OOP, Functions
- **Effect System:** Free/Freer, Tagless Final, ReaderT
- **OCaml User Defined Effects**

We picked this for the hype

HOW?

- **Trivial reason:** Unix, Windows, abstracting over the platform
- **Drastically facilitates testability** (a test-context is just another platform)
- Supports **exotic target types** (ie: GIT + Mirage)

WHY?

BUT

Execution abstraction

Grim's web corner

Notes, essays and ramblings

Basic dependency injection with objects

Published on 2025-08-18

In his article *Why I chose OCaml as my primary language*, my friend Xavier Van de Woestyne presents, in the section *Dependency injection and inversion*, two approaches to implementing dependency injection: one using user-defined effects and one using modules as first-class values. Even though I'm quite convinced that both approaches are *legit*, I find them sometimes a bit *overkill* and showing fairly obvious pitfalls when applied to real software. The goal of this article is theref

We define 12 effects

We define 12 effects

```
type filesystem = [ `Source | `Target ]
type _ Effect.t +=
  | Yocaml_log :
      (Logs.src option * [ `App | `Error | `Warning | `Info | `Debug ] * string)
      -> unit Effect.t
  | Yocaml_failwith : exn -> 'a Effect.t
  | Yocaml_get_time : unit -> int Effect.t
  | Yocaml_file_exists : filesystem * Path.t -> bool Effect.t
  | Yocaml_read_file : filesystem * bool * Path.t -> string Effect.t
  | Yocaml_get_mtime : filesystem * Path.t -> int Effect.t
  | Yocaml_hash_content : string -> string Effect.t
  | Yocaml_write_file : filesystem * Path.t * string -> unit Effect.t
  | Yocaml_is_directory : filesystem * Path.t -> bool Effect.t
  | Yocaml_read_dir : filesystem * Path.t -> Path.fragment list Effect.t
  | Yocaml_create_dir : filesystem * Path.t -> unit Effect.t
  | Yocaml_exec_command :
      string * string list * (int -> bool)
      -> string Effect.t
```

We define 12 effects

To distinguish between the target and the source (particularly useful for Git)

```
type filesystem = [ `Source | `Target ]
type _ Effect.t +=
  | Yocaml_log :
      (Logs.src option * [ `App | `Error | `Warning | `Info | `Debug ] * string)
      -> unit Effect.t
  | Yocaml_failwith : exn -> 'a Effect.t
  | Yocaml_get_time : unit -> int Effect.t
  | Yocaml_file_exists : filesystem * Path.t -> bool Effect.t
  | Yocaml_read_file : filesystem * bool * Path.t -> string Effect.t
  | Yocaml_get_mtime : filesystem * Path.t -> int Effect.t
  | Yocaml_hash_content : string -> string Effect.t
  | Yocaml_write_file : filesystem * Path.t * string -> unit Effect.t
  | Yocaml_is_directory : filesystem * Path.t -> bool Effect.t
  | Yocaml_read_dir : filesystem * Path.t -> Path.fragment list Effect.t
  | Yocaml_create_dir : filesystem * Path.t -> unit Effect.t
  | Yocaml_exec_command :
      string * string list * (int -> bool)
      -> string Effect.t
```

But effects **are not tracked** in the type system!

We abstract the execution that can be implemented with an **Effect Handler**.

But effects **are not tracked** in the type system!

We abstract the execution that can be implemented with an **Effect Handler**.

But effects **are not tracked** in the type system!

Without typing, the effect's performance can **be leaked**.
This forces us to be **overly cautious**.

We abstract the execution that can be implemented with an **Effect Handler**.

But effects **are not tracked** in the type system!

Without typing, the effect's performance can **be leaked**.
This forces us to be **overly cautious**.

So let's add a *naive typing*, "**may or may not produce an effect**" with an **IO monad**.

Here is our Eff module

```ocaml
type 'a t = unit -> 'a

let return x () = x
let bind f x = f (x ())
let map f x = bind (fun m -> return @@ f m) x
let apply ft xt = map (ft ()) xt
let zip x y = apply (map (fun a b -> (a, b)) x) y

let perform raw_effect =
   return @@ Effect.perform raw_effect

module Syntax = struct
 let ( let+ ) x f = map f x
 let ( and+ ) = zip
 let ( let* ) x f = bind f x
end

let file_exists ~on path =
  perform @@ Yocaml_file_exists (on, path)
```

Here is our Eff module

Usual combinators
(for Functor, Applicative and Monad)

```
type 'a t = unit -> 'a

let return x () = x
let bind f x = f (x ())
let map f x = bind (fun m -> return @@ f m) x
let apply ft xt = map (ft ()) xt
let zip x y = apply (map (fun a b -> (a, b)) x) y

let perform raw_effect =
   return @@ Effect.perform raw_effect

module Syntax = struct
 let ( let+ ) x f = map f x
 let ( and+ ) = zip
 let ( let* ) x f = bind f x
end

let file_exists ~on path =
  perform @@ Yocaml_file_exists (on, path)
```

Here is our Eff module

```ocaml
type 'a t = unit -> 'a

let return x () = x
let bind f x = f (x ())
let map f x = bind (fun m -> return @@ f m) x
let apply ft xt = map (ft ()) xt
let zip x y = apply (map (fun a b -> (a, b)) x) y

let perform raw_effect =
   return @@ Effect.perform raw_effect

module Syntax = struct
 let ( let+ ) x f = map f x
 let ( and+ ) = zip
 let ( let* ) x f = bind f x
end

let file_exists ~on path =
  perform @@ Yocaml_file_exists (on, path)
```

Usual combinators
(for Functor, Applicative and Monad)

since **'a t** is abstract, we can lift
a **'a Effect.t** to a **'a t**

Here is our Eff module

```ocaml
type 'a t = unit -> 'a

let return x () = x
let bind f x = f (x ())
let map f x = bind (fun m -> return @@ f m) x
let apply ft xt = map (ft ()) xt
let zip x y = apply (map (fun a b -> (a, b)) x) y

let perform raw_effect =
   return @@ Effect.perform raw_effect

module Syntax = struct
 let ( let+ ) x f = map f x
 let ( and+ ) = zip
 let ( let* ) x f = bind f x
end

let file_exists ~on path =
  perform @@ Yocaml_file_exists (on, path)
```

Usual combinators
(for Functor, Applicative and Monad)

since **'a t** is abstract, we can lift
a **'a Effect.t** to a **'a t**

we simplify usage with
binding operators

Here is our Eff module

```
type 'a t = unit -> 'a

let return x () = x
let bind f x = f (x ())
let map f x = bind (fun m -> return @@ f m) x
let apply ft xt = map (ft ()) xt
let zip x y = apply (map (fun a b -> (a, b)) x) y

let perform raw_effect =
   return @@ Effect.perform raw_effect

module Syntax = struct
 let ( let+ ) x f = map f x
 let ( and+ ) = zip
 let ( let* ) x f = bind f x
end

let file_exists ~on path =
  perform @@ Yocaml_file_exists (on, path)
```

Usual combinators
(for Functor, Applicative and Monad)

since **'a t** is abstract, we can lift
a **'a Effect.t** to a **'a t**

we simplify usage with
binding operators

And we can wrap all our effects with
perform

Here is our Eff module

```ocaml
type 'a t = unit -> 'a

let return x () = x
let bind f x = f (x ())
let map f x = bind (fun m -> return @@ f m) x
let apply ft xt = map (ft ()) xt
let zip x y = apply (map (fun a b -> (a, b)) x) y
```

Usual combinators
(for Functor, Applicative and Monad)

```ocaml
let perform raw_effect =
  return @@ Effect.perform raw_effect
```

since **'a t** is abstract, we can lift a **'a Effect.t** to a **'a t**

```ocaml
module Syntax = struct
 let ( let+ ) x f = map f x
 let ( and+ ) = zip
 let ( let* ) x f = bind f x
end
```

we simplify usage with **binding operators**

```ocaml
let file_exists ~on path =
  perform @@ Yocaml_file_exists (on, path)
```

And we can wrap all our effects with **perform**

```ocaml
let sample () =
 let* exists =
   file_exists
     ~on:`Source Path.root
 in
 if exists then
   log "File exists"
 else log "File does not exists"
```

We lose the direct style , but we can distinguish between *pure and impure* **computation** .

We can interpret our “**a Eff.t**” type programmes by applying our calculation (with a **run** function that simply passes () to an expression).

So abstraction over the execution is mostly fixed.

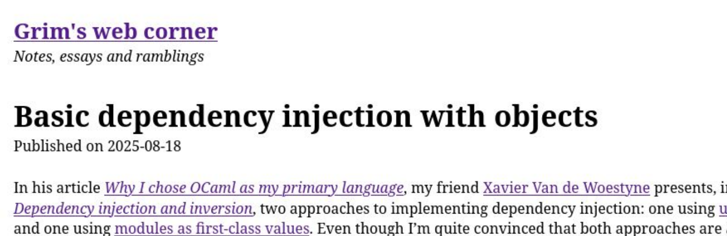
We lose the direct style, but we can distinguish between *pure and impure* **computation**.

We can interpret our "**a Eff.t**" type programmes by applying our calculation (with a **run** function that simply passes () to an expression).

So abstraction over the execution is mostly fixed.

We lose the direct style, but we can distinguish between *pure and impure* computation.

Since we do not really handle the continuation effect are probably too much


Grim's web corner
Notes, essays and ramblings

Basic dependency injection with objects

Published on 2025-08-18

In his article *Why I chose OCaml as my primary language*, my friend Xavier Van de Woestyne presents, i *Dependency injection and inversion*, two approaches to implementing dependency injection: one using u and one using modules as first-class values. Even though I'm quite convinced that both approaches are

We can interpret our “**a Eff.t**” type programmes by applying our calculation (with a **run** function that simply passes () to an expression).

So abstraction over the execution is mostly fixed.

We lose the direct style, but we can distinguish between *pure and impure* **computation**.

Since we do not really handle the continuation effect are probably too much

Grim's web corner

Notes, essays and ramblings

Basic dependency injection with objects

Published on 2025-08-18

In his article *Why I chose OCaml as my primary language*, my friend Xavier Van de Woestyne presents, i *Dependency injection and inversion*, two approaches to implementing dependency injection: one using u and one using modules as first-class values. Even though I'm quite convinced that both approaches are

Let's move to dealing with **metadata validation**

There are **hundreds** of metadata description languages.

Yaml, ToML, Json,
Sexp, etc.

There are **hundreds** of metadata description languages.

And there are **hundreds of Templates languages**

Yaml, ToML, Json, Sexp, etc.

There are **hundreds** of metadata description languages.

And there are **hundreds of Templates languages**

Yaml, ToML, Json, Sexp, etc.

There are **hundreds** of metadata description languages.

And we probably **don't want** to lock our potential users into **a choice that is set in stone**

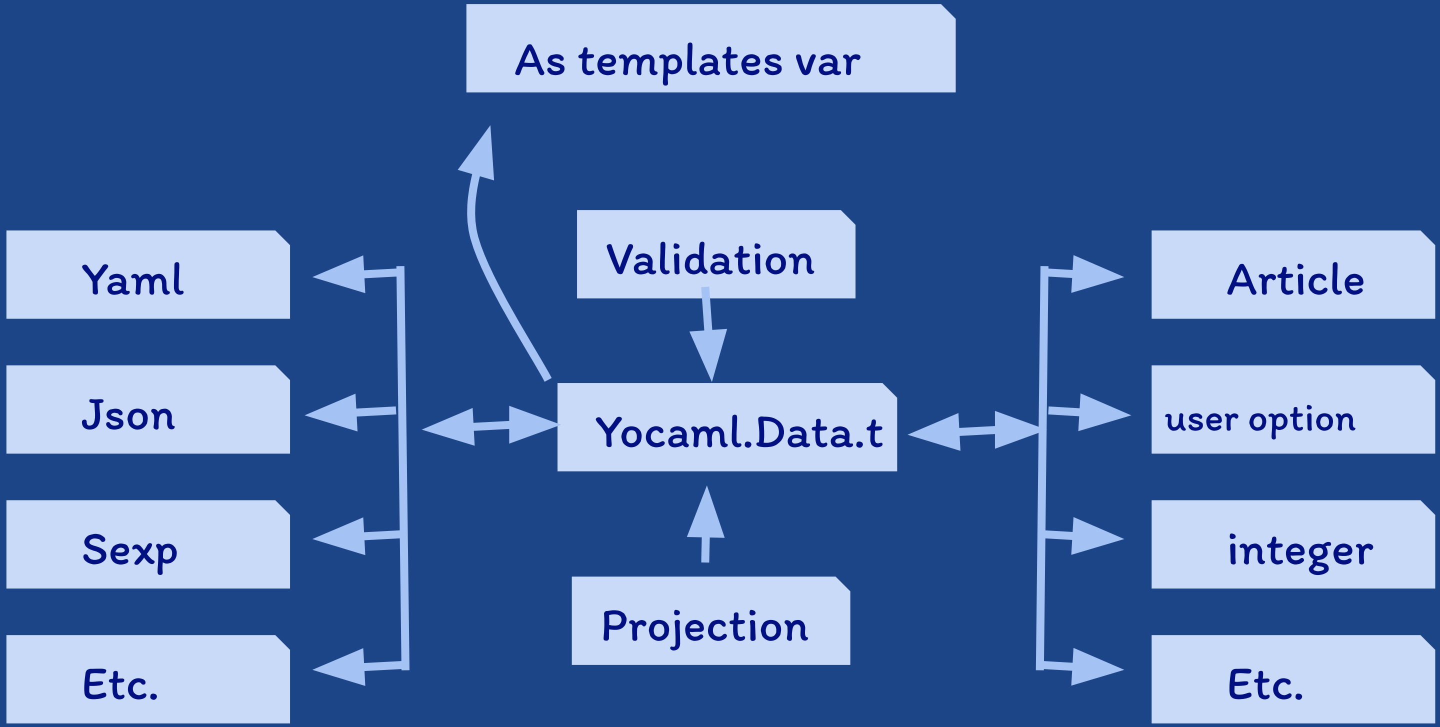
And there are **hundreds of Templates languages**

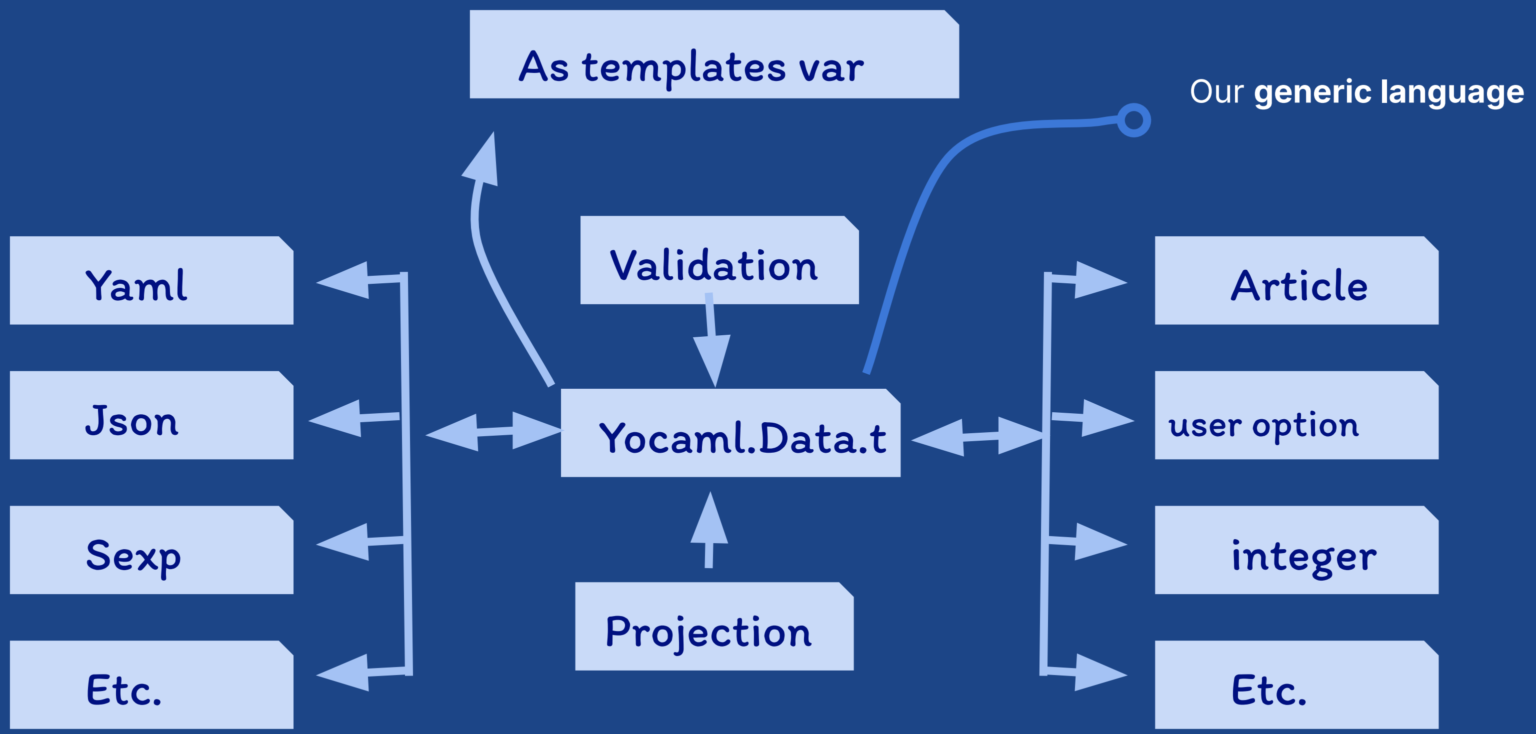
Yaml, ToML, Json, Sexp, etc.

There are **hundreds** of metadata description languages.

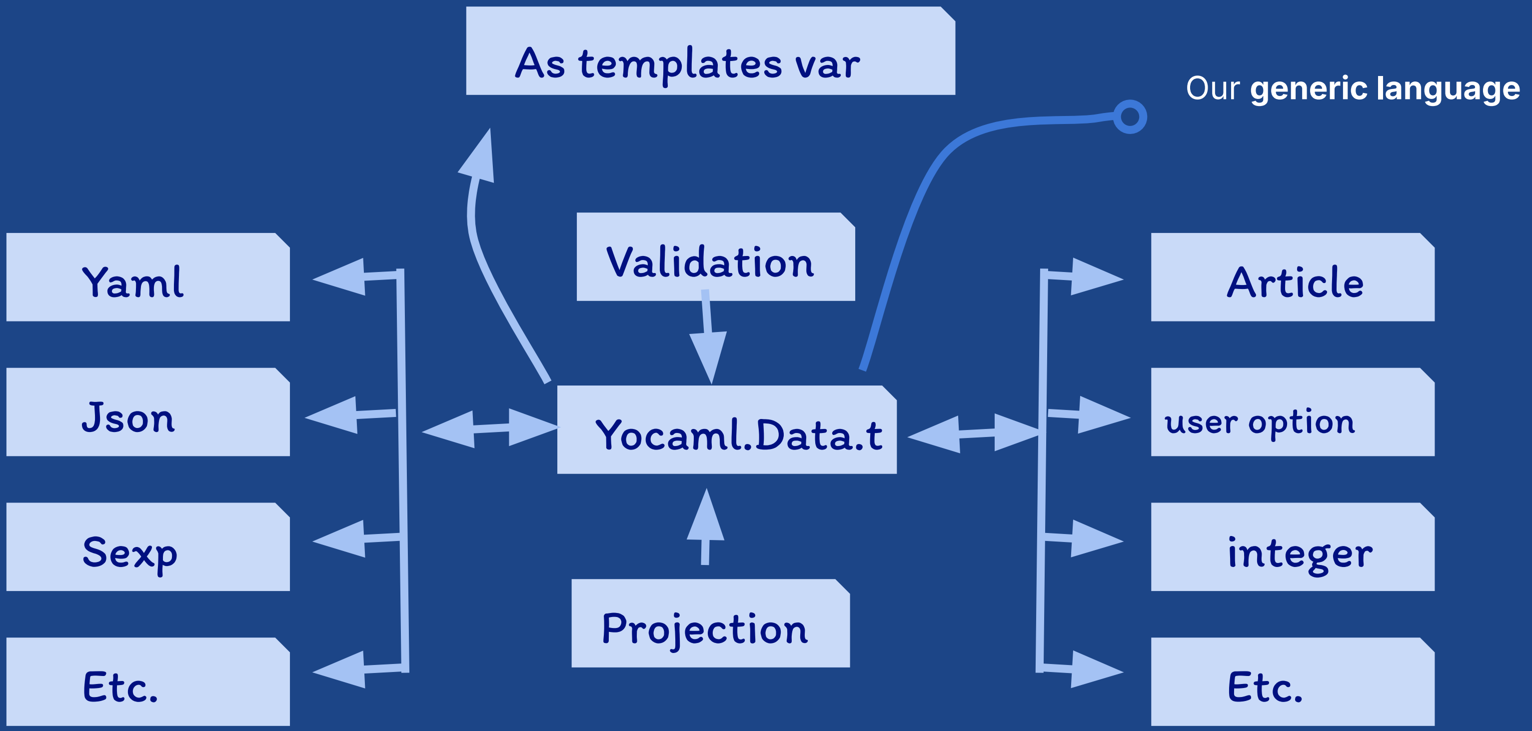
And we probably **don't want** to lock our potential users into **a choice that is set in stone**

let's abstract the notion of **Key-Value** language (in a naive way)


As templates var
Yaml
Json
Sexp
Etc.
Validation
Yocaml.Data.t
Projection
Article
user option
integer
Etc.



As templates var
Our **generic language**
Validation
Yaml
Json
Sexp
Etc.
Yocaml.Data.t
Projection
Article
user option
integer
Etc.



Even though intermediate representation introduces indirection, we believe it is worthwhile (as opposed to a module-based implementation).

A very simple AST that can describe many things!

```
type t = private
  | Null
  | Bool of bool
  | Int of int
  | Float of float
  | String of string
  | List of t list
  | Record of (string * t) list
```

```
val null : t
val bool : bool -> t
val int : int -> t
val float : float -> t
val string : string -> t
val list : t list -> t
val list_of : ('a -> t) -> 'a list -> t
val record : (string * t) list -> t
val option : ('a -> t) -> 'a option -> t
val sum : ('a -> string * t) -> 'a -> t
val pair : ('a -> t) -> ('b -> t) -> 'a * 'b -> t
val triple :
 ('a -> t) -> ('b -> t)
 -> ('c -> t) -> 'a * 'b * 'c -> t
val quad :
 ('a -> t) -> ('b -> t)
 -> ('c -> t) -> ('d -> t)
 -> 'a * 'b * 'c * 'd -> t
 val either :
('a -> t) -> ('b -> t)
-> ('a, 'b) Either.t -> t
```

A very simple AST that can describe many things!

```
type t = private
  | Null
  | Bool of bool
  | Int of int
  | Float of float
  | String of string
  | List of t list
  | Record of (string * t) list
```

```
val null : t
val bool : bool -> t
val int : int -> t
val float : float -> t
val string : string -> t
val list : t list -> t
val list_of : ('a -> t) -> 'a list -> t
val record : (string * t) list -> t
val option : ('a -> t) -> 'a option -> t
val sum : ('a -> string * t) -> 'a -> t
val pair : ('a -> t) -> ('b -> t) -> 'a * 'b -> t
val triple :
 ('a -> t) -> ('b -> t)
 -> ('c -> t) -> 'a * 'b * 'c -> t
val quad :
 ('a -> t) -> ('b -> t)
 -> ('c -> t) -> ('d -> t)
 -> 'a * 'b * 'c * 'd -> t
 val either :
('a -> t) -> ('b -> t)
-> ('a, 'b) Either.t -> t
```

And we associate **definition** with **validation**

A very simple AST that can describe many things!

```ocaml
type t = private
  | Null
  | Bool of bool
  | Int of int
  | Float of float
  | String of string
  | List of t list
  | Record of (string * t) list
```

In practice, even though the API could be refined, it **seems sufficient** (hence the success of JSON).

```ocaml
val null : t
val bool : bool -> t
val int : int -> t
val float : float -> t
val string : string -> t
val list : t list -> t
val list_of : ('a -> t) -> 'a list -> t
val record : (string * t) list -> t
val option : ('a -> t) -> 'a option -> t
val sum : ('a -> string * t) -> 'a -> t
val pair : ('a -> t) -> ('b -> t) -> 'a * 'b -> t
val triple :
 ('a -> t) -> ('b -> t)
 -> ('c -> t) -> 'a * 'b * 'c -> t
val quad :
 ('a -> t) -> ('b -> t)
 -> ('c -> t) -> ('d -> t)
 -> 'a * 'b * 'c * 'd -> t
 val either :
('a -> t) -> ('b -> t)
-> ('a, 'b) Either.t -> t
```

And we associate **definition** with **validation**

```ocaml
let validate_article =
 let open Yocaml.Data.Validation in
 record (fun fields ->
     let+ title = required fields "title"        string
     and+ desc  = optional fields "description"  string
     and+ date  = required fields "date"         Datetime.validate
     in make_article title desc date
   )
```

```ocaml
let validate_article =
 let open Yocaml.Data.Validation in
 record (fun fields ->
     let+ title = required fields "title"       string
     and+ desc  = optional fields "description" string
     and+ date  = required fields "date"        Datetime.validate
     in make_article title desc date
   )
```

```ocaml
(string * Data.t) list
   -> string -> 'a Data.validator ->
   -> ('a, SEMIGROUP) Result.t
```

for **collecting all errors**

```ocaml
let validate_article =
 let open Yocaml.Data.Validation in
 record (fun fields ->
     let+ title = required fields "title"          string
     and+ desc  = optional fields "description"  string
     and+ date  = required fields "date"          Datetime.validate
     in make_article title desc date
   )
```

```ocaml
Data.t ->
  ((string * Data.t) list ->
     'a validated_record) ->
  'a validated_value
```

record is a regular **validator.**

```
let validate_article =
 let open Yocaml.Data.Validation in
 record (fun fields ->
     let+ title = required fields "title"          string
     and+ desc  = optional fields "description"    string
     and+ date  = required fields "date"           Datetime.validate
     in make_article title desc date
   )
```

```
Data.t -> 'a validator
-> 'a validated_value
```

- **Fields are apply in parallel**
- **validators inside fields are sequentials**
  (and hold Alternative)

Article.t validator
(Data.t -> Article.t validated_value)

```
let validate_article =
 let open Yocaml.Data.Validation in
 record (fun fields ->
     let+ title = required fields "title"           string
     and+ desc  = optional fields "description"  string
     and+ date  = required fields "date"            Datetime.validate
     in make_article title desc date
   )
```

So **we can use validate_article as an other field validator**

```
Article.t validator
(Data.t -> Article.t validated_value)
```

```
let validate_article =
 let open Yocaml.Data.Validation in
 record (fun fields ->
     let+ title = required fields "title"          string
     and+ desc  = optional fields "description"  string
     and+ date  = required fields "date"           Datetime.validate
     in make_article title desc date
   )
```

This gives us a very good insight into the nuance between Applicative and monad.

This gives us a very good insight into the nuance between Applicative and monad.

Can perform **parallel task** (allowing static analysis without dry-run)

This gives us a very good insight into the nuance between Applicative and monad.

Can perform **parallel task** (allowing static analysis without dry-run)

Can perform **sequential task** (and dynamic behaviour)

This gives us a very good insight into the nuance between Applicative and monad.

Can perform **parallel task** (allowing static analysis without dry-run)

Can perform **sequential task** (and dynamic behaviour)

We can use Monad to **handle pre-condition** in validation (and following by an applicative pipeline)

Let's talk about **minimality**

This gives us a very good insight into the nuance between Applicative and monad.

Can perform **parallel task** (allowing static analysis without dry-run)

Can perform **sequential task** (and dynamic behaviour)

We can use Monad to **handle pre-condition** in validation (and following by an applicative pipeline)

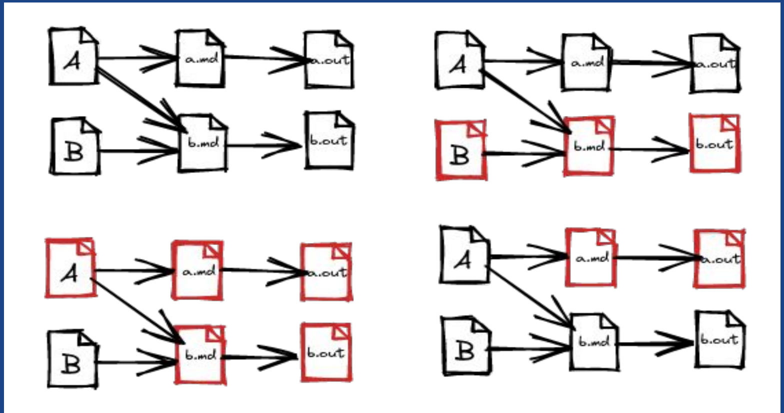
Minimality

Minimality

Attempt not to perform tasks that do not need to be performed

Minimality

Attempt not to perform tasks that do not need to be performed



```
let () =
 let file = Sys.argv.(1) in
 let file_html =
   file
   |> Filename.basename
   |> Filename.remove_extension
 in
 let target = "_www/" ^ file_html in
 let (metadata, content) = File.read file in
 let markdown = Markdown.of_string content in
 let injected =
   Template.inject
   "article.html"
   metadata
   markdown
 in File.write target injected
```

```ocaml
let () =
 let file = Sys.argv.(1) in
 let file_html =
   file
   |> Filename.basename
   |> Filename.remove_extension
 in
 let target = "_www/" ^ file_html in
 let (metadata, content) = File.read file in
 let markdown = Markdown.of_string content in
 let injected =
   Template.inject
   "article.html"
   metadata
   markdown
 in File.write target injected
```

We can split the program in 5 task:

- **compute the target**
- **read the file**
- **convert it**
- **inject it using article.html**
- **write the file target**

```ocaml
let () =
 let file = Sys.argv.(1) in
 let file_html =
   file
   |> Filename.basename
   |> Filename.remove_extension
 in
 let target = "_www/" ^ file_html in
 let (metadata, content) = File.read file in
 let markdown = Markdown.of_string content in
 let injected =
   Template.inject
   "article.html"
   metadata
   markdown
 in File.write target injected
```

We can split the program in 5 task:

- **compute the target**
- **read the file**
- **convert it**
- **inject it using article.html**
- **write the file target**

So the Create File action as 3 task

- **read the file**
- **convert it**
- **inject it using article.html**

And has **2 static dependencies:**

- the file (to be read)
- article.html

```ocaml
let () =
 let file = Sys.argv.(1) in
 let file_html =
   file
   |> Filename.basename
   |> Filename.remove_extension
 in
 let target = "_www/" ^ file_html in
 let (metadata, content) = File.read file in
 let markdown = Markdown.of_string content in
 let injected =
   Template.inject
   "article.html"
   metadata
   markdown
 in File.write target injected
```

We can split the program in 5 task:

- **compute the target**
- **read the file**
- **convert it**
- **inject it using article.html**
- **write the file target**

So the Create File action as 3 task

- **read the file**
- **convert it**
- **inject it using article.html**

And has **2 static dependencies:**

- the file (to be read)
- article.html

So **we need to build the target IF:**

- target does not exists
  OR:
  mtime(target) <
  max(mtime(source), mtime(target))

That leads to:

```
type (-'in, +'out) task
val create_file : Path.t -> (unit, string) task -> unit
```

That leads to:

```
type (-'in, +'out) task
val create_file : Path.t -> (unit, string) task -> unit
```

where **create_file performs the given task only if it respect the previous heuristic.**

That leads to:

Let's define task!

```
type (-'in, +'out) task
val create_file : Path.t -> (unit, string) task -> unit
```

where **create_file performs the given task only if it respect the previous heuristic.**

```ocaml
type ('a, 'b) task = {
 action: ('a -> 'b Eff.t)
; deps: Deps.t (* A Set of Path*)
}

let run {action; _} x =
 action x

let lift f = {
 action = (fun x -> Eff.return (f x))
; deps = Deps.empty
}
```

```ocaml
let track_file file = {
 action = Eff.return
; deps = Deps.singleton file
}
```

```
type ('a, 'b) task = {
 action: ('a -> 'b Eff.t)
; deps: Deps.t (* A Set of Path*)
}

let run {action; _} x =
 action x

let lift f = {
 action = (fun x -> Eff.return (f x))
; deps = Deps.empty
}
```

```
let track_file file = {
 action = Eff.return
; deps = Deps.singleton file
}
```

We can also imagine **reading a file**

```
let read_file file = {
 action = (fun () ->
   Eff.read_file ~on:`Source file)
; deps = Deps.singleton file
}
```

But hey, we could use `track_file` in `read_file` no?

How to compose task?

But hey, we could use `track_file` in `read_file` no?

How to compose task?

```
let (>>>) t1 t2 =
 let deps = Deps.concat t1.deps t2.deps in
 let action x =
   let open Eff.Syntax in
   let* y = run t1 x in
   run t2 y
 in
 {action; deps}
```

At this stage, task looks like a function, but it is not!

At this stage, task looks like a function, but it is not!

we can sequentially compose **task:**
t1 >>> t2

That will collect statically
dependencies and produce a new task
that will **perform t1 following by t2**

But in fact, **by the magic of higher kinded abstraction** we **can generate a lot of combinators**

At this stage, task looks like a function, but it is not!

we can sequentially compose **task:**
t1 >>> t2

That will collect statically dependencies and produce a new task that will **perform t1 following by t2**

In fact, task is **a semigroupoid associated with a profunctor with a strength tensor** 😂

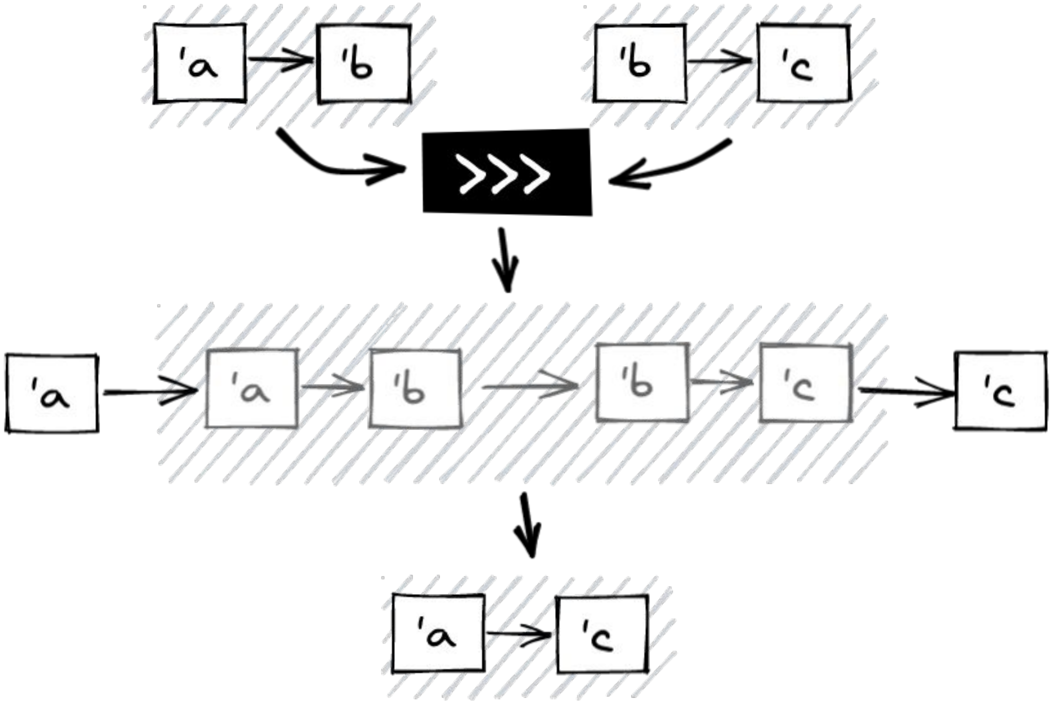
(also called … an Arrow)

But in fact, **by the magic of higher kinded abstraction** we **can generate a lot of combinators**

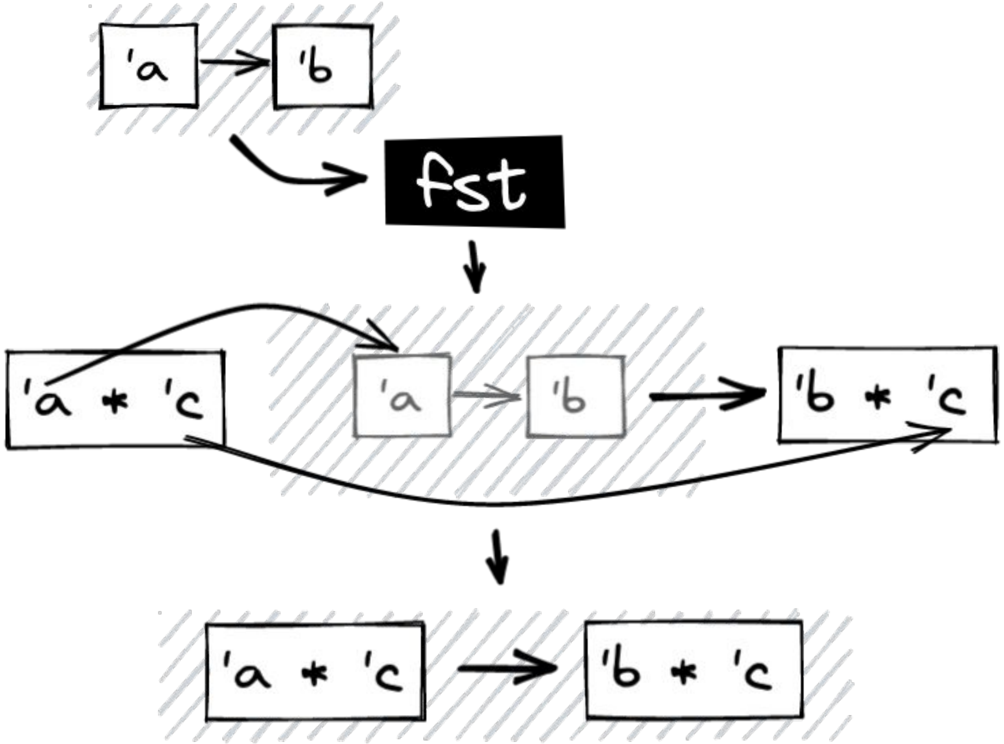
At this stage, task looks like a function, but it is not!

we can sequentially compose **task:**
t1 >>> t2

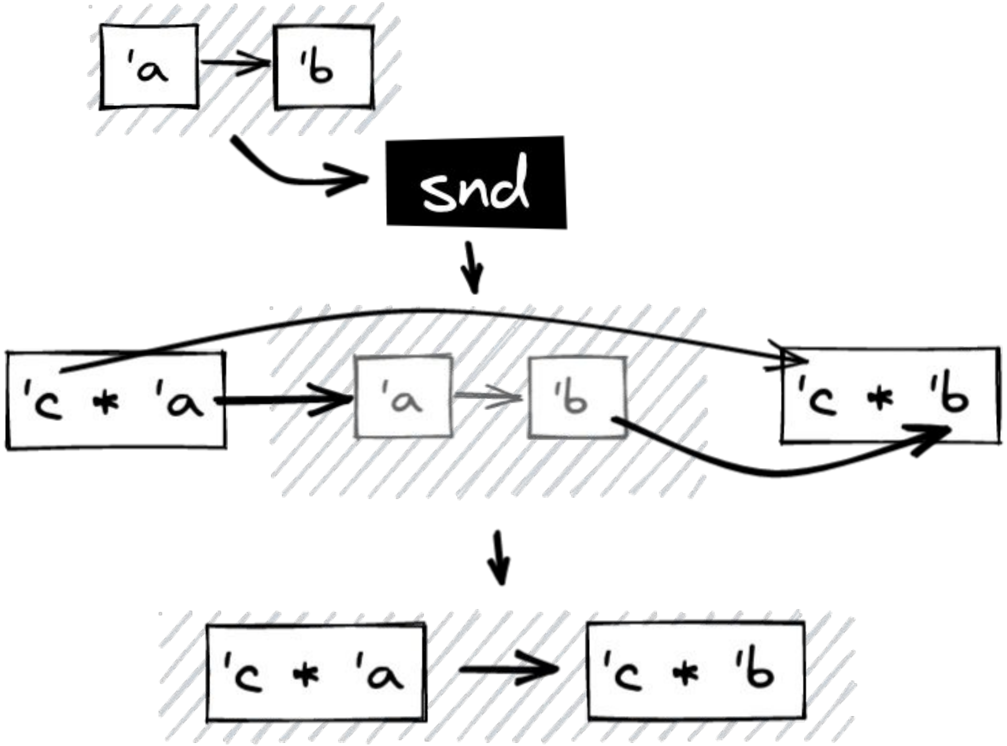
That will collect statically dependencies and produce a new task that will **perform t1 following by t2**

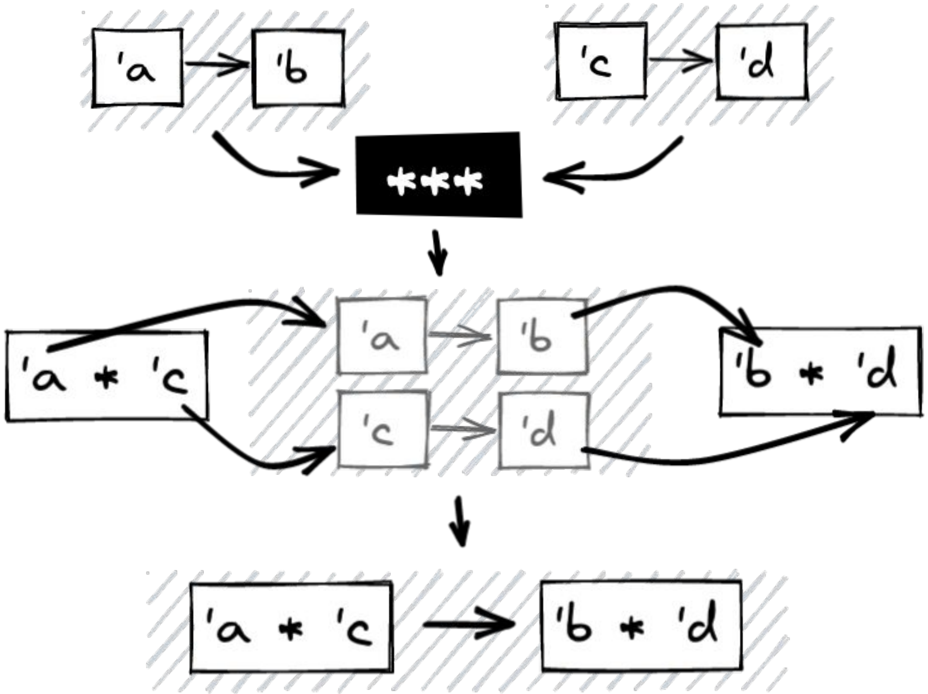


'a
'b
'b
'c
>>>
'a
'a
'b
'b
'c
'c
'a
'c

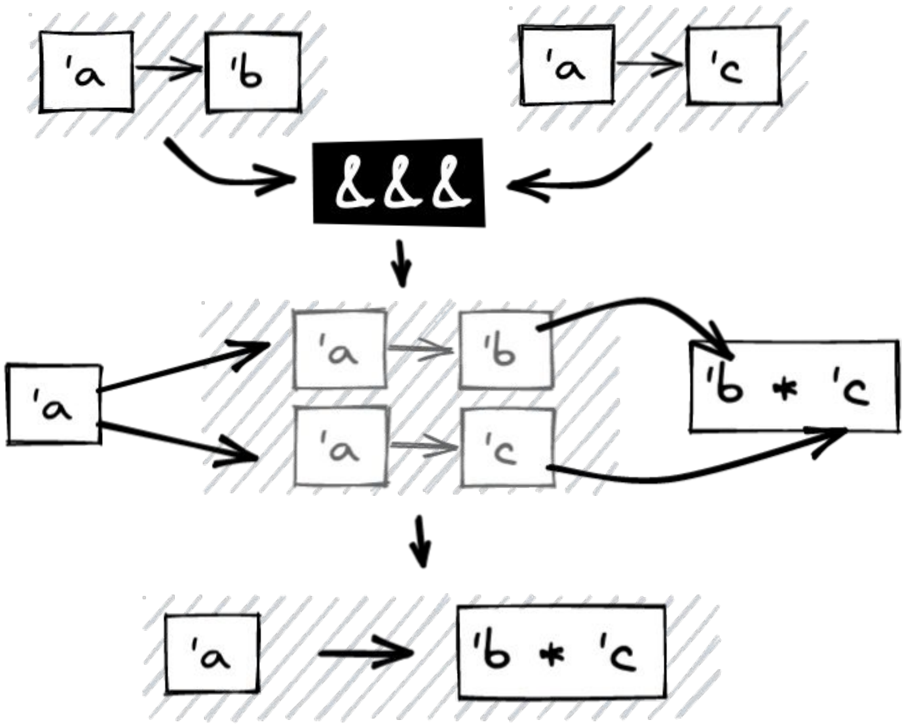


'a
'b
fst
'a * 'c
'a
'b
'b * 'c
'a * 'c
'b * 'c


'a
'b
snd
'c * 'a
'a
'b
'c * 'b
'c * 'a
'c * 'b

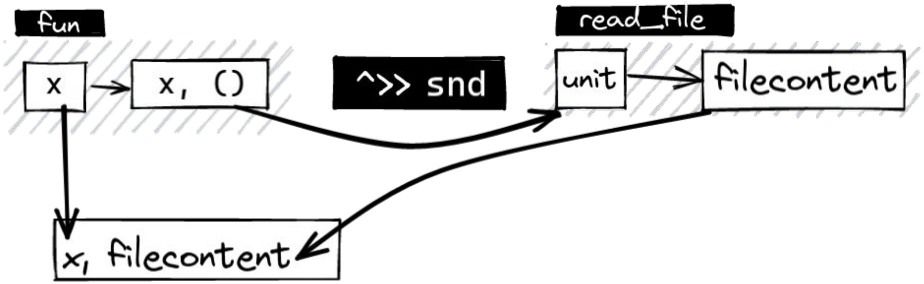


'a
'b
'c
'd
***
'a
'b
'c
'd
'a * 'c
'b * 'd
'a * 'c
'b * 'd


'a
'b
'a
'c
&&&
'a
'a
'b
'a
'c
'b * 'c
'a
'b * 'c

All these combinators require us to programme in pointfree, but they give us a great deal of control to create increasingly complex tasks.

An example of task that can pipe files




```
let pipe_content filename =
   lift (fun x -> x, ())
   >> snd (read_file filename)
   >> lift (fun (content_a, content_b) -> content_a ^ content_b)
```

In practice, we mostly use >>>, fst and snd.

In practice, we mostly use >>>, fst and snd.

```
let process_page file =
  let file_target = Target.(as_html pages file) in
  let open Task in
  Action.Static.write_file_with_metadata file_target
    (Pipeline.track_file Source.binary
     >>> Yocaml_yaml.Pipeline.read_file_with_metadata
           (module Archetype.Page)
           file
     >>> Yocaml_omd.content_to_html ()
     >>> Yocaml_jingoo.Pipeline.as_template
           (module Archetype.Page)
           (Source.template "page.html")
     >>> Yocaml_jingoo.Pipeline.as_template
           (module Archetype.Page)
           (Source.template "layout.html"))
```

this is **real world code !**

The key idea:

collecting dependencies **before** executing the action, and building an action **by composition**

But hey, **we hate pointfree style** !

But hey, **we hate pointfree style** !

Yes, that's fair! We also have
an applicative API !

```ocaml
let create_page source =
 let page_path =
   source
   |> Path.move ~into:www
   |> Path.change_extension "html"
 in
 let pipeline =
   let open Task in
   let+ () = track_binary
   and+ apply_templates =
      Yocaml_jingoo.read_templates
        Path.[ templates / "page.html"
             ; templates / "layout.html" ]
   and+ metadata, content =
     Yocaml_yaml.Pipeline.read_file_with_metadata
       (module Archetype.Page)
       source
   in
   content
   |> Yocaml_markdown.from_string_to_html
   |> apply_templates (module Archetype.Page) ~metadata
 in
 Action.Static.write_file page_path pipeline
```

```ocaml
let create_page source =
 let page_path =
   source
   |> Path.move ~into:www
   |> Path.change_extension "html"
 in
 let pipeline =
   let open Task in
   let+ () = track_binary
   and+ apply_templates =
      Yocaml_jingoo.read_templates
        Path.[ templates / "page.html"
             ; templates / "layout.html" ]
   and+ metadata, content =
     Yocaml_yaml.Pipeline.read_file_with_metadata
       (module Archetype.Page)
       source
   in
   content
   |> Yocaml_markdown.from_string_to_html
   |> apply_templates (module Archetype.Page) ~metadata
 in
 Action.Static.write_file page_path pipeline
```

under the hood, it still a task: **(unit, 'a) task is an 'a applicative.**

But it is more usable.

but sometimes **Arrows give more control** , especially when you want to compute a state **that depends on the previous task** .

The YOCaml API is much richer than what we have seen and offers a **complete DSL for building static websites**! I really encourage you to try it out because it's a lot of fun!

The YOCaml API is much richer than what we have seen and offers a **complete DSL for building static websites**! I really encourage you to try it out because it's a lot of fun!

- https://github.com/xhtmlboi
- https://yocaml.github.io/doc
- https://yocaml.github.io/tutorial

But YOCaml **offers advantages**

Now that you understand the key points behind YOCaml, why use it and **not reinvent the wheel** ?

Please, **do it!** **It's so cool to have alternatives**

It's **maintained**

and used by users other than maintainers

It's **well documented**

API Doc, Guides and Examples

A lot of plugin based on

popular libraries

Markdown, Mustache, Templates, RSS/Atom,

Syntax Highlighting, Git

Features not covered

Dynamic Dependencies, Caches, Snapshots

But the most important part: **with or without YOCaml**, maintain your own websites! Less Medium! More personal websites

(**and ideally implemented in OCaml**)

The End

Question, Remarks ?

I would be delighted to discuss dynamic dependencies with you privately in order to approximate the functionality of Xanadu in the Kane project (based on YOCaml).