les **systèmes de construction** (*build system*)

Théorie informelle et pratique

Xavier Van de Woestyne - xvw.lol

Un build system est un logiciel (ou une collection de logiciels) permettant d'automatiser certaines tâches récurrentes. Par exemple produire des artefacts (compilation, unités, typechecking, documentation), exécuter des tests, linter.

Pourquoi s'y intéresser ?

- Ce sont des logiciels invasifs dans notre métier que l'on comprend souvent mal ;
- je trouve que c'est un sujet intéressant (pour lequel l'approche théorique est récente) ;
- ça permet de comprendre les limites (et les *trade off*) de certains (`cabal`, `dune + opam`, `maven`) ;
- diminués, ils permettent de solutionner des problèmes usuels (par exemple, décrire un générateur de blogs statiques).

Plan

- Origine de mon intérêt pour les *build systems* ;
- Une brève frise chronologique ;
- Implémentations et considérations
- Conclusions

Un intérêt subite grâce à Preface !

Preface is an opinionated library designed to facilitate the handling of recurring functional programming idioms in OCaml.

Par Didier Plaindoux, Pierre Ruyter, Michaël Spawn, moi et d'autres contributeurs occasionnels

- Une bibliothèque *pour programmer comme en Haskell*
- conçue à des fins pédagogiques
- après un premier usage sur un projet personnel

Un intérêt subite grâce à Preface !

Preface is an opinionated library designed to facilitate the handling of recurring functional programming idioms in OCaml.

Par Didier Plaindoux, Pierre Ruyter, Michaël Spawn, moi et d'autres contributeurs occasionnels

- Une bibliothèque *pour programmer comme en Haskell*
- conçue à des fins pédagogiques
- après un premier usage sur un projet personnel
- **inutilisable sur beaucoup d'aspects**

Un intérêt subite grâce à Preface !

> *Preface is an opinionated library designed to facilitate the handling of recurring functional programming idioms in OCaml.*
>
> *Par Didier Plaindoux, Pierre Ruyter, Michaël Spawn, moi et d'autres contributeurs occasionnels*

- Une bibliothèque *pour programmer comme en Haskell*
- conçue à des fins pédagogiques
- après un premier usage sur un projet personnel
- **inutilisable sur beaucoup d'aspects**

C'est pour ça que **YOCaml** a été conçu: **expérimenter l'expérience utilisateur** de Preface avec un projet qui utilise des abstractions un peu moins populaires dans *le monde des blog-posts*.

Ce qui explique parfois **des choix relativement peu pragmatiques**.

Ce qui explique parfois **des choix relativement peu pragmatiques**.

Parce que pour **expérimenter l'expérience utilisateur** de Preface, nous avons conçu un outil à **l'expérience utilisateur douteuse**...

Ce qui explique parfois **des choix relativement peu pragmatiques**.

Parce que pour **expérimenter l'expérience utilisateur** de Preface, nous avons conçu un outil à **l'expérience utilisateur douteuse**...

Mais après un an, YOCaml est un peu utilisé et certains (dont moi) en sont même satisfait ! Et YOCaml est un **tout petit build system** qui utilise un encodage élégant à base de *Profunctors*.

Au même moment

Sortie des papiers suivants:

Build systems à la carte

Par **A. Mokhov**, N. Mitchell, S. P. Jones

Une tentative convaincante de formalisation des Build Systems avec une souche théorique robuste.

Au même moment

Sortie des papiers suivants:

Build systems à la carte

Par **A. Mokhov**, N. Mitchell, S. P. Jones

Une tentative convaincante de formalisation des Build Systems avec une souche théorique robuste.

Selective Applicative Functors

Par **A. Mokhov**, G. Lukyanov, S. Marlow, J. Dimino

L'ajout d'une abstraction entre Applicative et Monad qui complète la hiérarchie des abstractions de kind d'artité 1 (et qui fournit une contrepartie à Arrow Choice).

Au même moment

Sortie des papiers suivants:

Build systems à la carte

Par **A. Mokhov**, N. Mitchell, S. P. Jones

Une tentative convaincante de formalisation des Build Systems avec une souche théorique robuste.

Selective Applicative Functors

Par **A. Mokhov**, G. Lukyanov, S. Marlow, J. Dimino

L'ajout d'une abstraction entre Applicative et Monad qui complète la hiérarchie des abstractions de kind d'artité 1 (et qui fournit une contrepartie à Arrow Choice).

Mokhov a été un relecteur consciencieux de la partie Selective (et Free/Freer selective) de Preface.

Sur l'utilisation des la théorie des catégories

Schéma sarcastique :

- **Fresh Haskell Programmer** : ne comprends rien à la théorie des catégories et l'évite

Sur l'utilisation des la théorie des catégories

Schéma sarcastique :

- **Fresh Haskell Programmer** : ne comprends rien à la théorie des catégories et l'évite
- **Haskell Programmer** : ne comprends rien à la théorie des catégories et veut en mettre partout et trouve la programmation *tacite* jolie

Sur l'utilisation des la théorie des catégories

Schéma sarcastique :

- **Fresh Haskell Programmer** : ne comprends rien à la théorie des catégories et l'évite
- **Haskell Programmer** : ne comprends rien à la théorie des catégories et veut en mettre partout et trouve la programmation *tacite* jolie
- **Experimented Haskell Programmer** :capture les idées derrière la théorie des catégories et argue que ça ne sert à rien pour être un développeur (-> Boring Haskell)

Sur l'utilisation des la théorie des catégories

Schéma sarcastique :

- **Fresh Haskell Programmer** : ne comprends rien à la théorie des catégories et l'évite
- **Haskell Programmer** : ne comprends rien à la théorie des catégories et veut en mettre partout et trouve la programmation *tacite* jolie
- **Experimented Haskell Programmer** :capture les idées derrière la théorie des catégories et argue que ça ne sert à rien pour être un développeur (-> Boring Haskell)
- **Haskell Superstar** : utilise la théorie des catégories pour décrire des formalismes et potentiellement faire des optimisations non triviales.

Sur l'utilisation des la théorie des catégories

Schéma sarcastique :

- **Fresh Haskell Programmer** : ne comprends rien à la théorie des catégories et l'évite
- **Haskell Programmer** : ne comprends rien à la théorie des catégories et veut en mettre partout et trouve la programmation *tacite* jolie
- **Experimented Haskell Programmer** :capture les idées derrière la théorie des catégories et argue que ça ne sert à rien pour être un développeur (-> Boring Haskell)
- **Haskell Superstar** : utilise la théorie des catégories pour décrire des formalismes et potentiellement faire des optimisations non triviales.

Le dernier point est intéressant.

A Categorical Theory of Patches

Par S. Mimram, C. Di Giusto

Une approche catégorique de l'application de patches dans le contexte ***des systèmes de gestion de versions***

A Categorical Theory of Patches

Par S. Mimram, C. Di Giusto

Une approche catégorique de l'application de patches dans le contexte ***des systèmes de gestion de versions***

- Utilisés par **Darcs** et plus recemment par **Pijul**
- une approche **très convaincante**

Build Systems à la carte explore une approche similaire :

Build Systems à la carte explore une approche similaire :

- une mise à plat terminologique ;

Build Systems à la carte explore une approche similaire :

- une mise à plat terminologique ;
- une capture concrète des abstractions liées à un *Build System* ;

Build Systems à la carte explore une approche similaire :

- une mise à plat terminologique ;
- une capture concrète des abstractions liées à un *Build System* ;
- une classification des différents types de *Build System* ;

Build Systems à la carte explore une approche similaire :

- une mise à plat terminologique ;
- une capture concrète des abstractions liées à un *Build System* ;
- une classification des différents types de *Build System* ;
- des proposition catégoriques (moins lourdes que dans la théorie des patches) ;

Build Systems à la carte explore une approche similaire :

- une mise à plat terminologique ;
- une capture concrète des abstractions liées à un *Build System* ;
- une classification des différents types de *Build System* ;
- des proposition catégoriques (moins lourdes que dans la théorie des patches) ;

Ici je dois donner un peu d'informations sur les papiers "dit à la Carte" (désolé, je ne sais pas comment faire des speakers notes. ***Lol****)*

La théorie des catégories permet plusieurs choses :

La théorie des catégories permet plusieurs choses :

- programmer avec des mots savants ;

La théorie des catégories permet plusieurs choses :

- programmer avec des mots savants ;
- *over engineer* beaucoup de choses ;

La théorie des catégories permet plusieurs choses :

- programmer avec des mots savants ;
- *over engineer* beaucoup de choses ;
- fournir parfois des modèles de résolutions de problèmes élégants ;

La théorie des catégories permet plusieurs choses :

- programmer avec des mots savants ;
- *over engineer* beaucoup de choses ;
- fournir parfois des modèles de résolutions de problèmes élégants ;
- construire des intuitions et des formalismes sur des problèmes complexes.

Au début, après les scripts, il y eut `Make`

- **La genèse** : des collections de scripts pour automatiser ;

Au début, après les scripts, il y eut `Make`

- **La genèse** : des collections de scripts pour automatiser ;
- **1976** : Make (simple et utile)

Au début, après les scripts, il y eut `Make`

- **La genèse** : des collections de scripts pour automatiser ;
- **1976** : Make (simple et utile)

```
%.x : %.y
  task list
```

Au début, après les scripts, il y eut `Make`

- **La genèse** : des collections de scripts pour automatiser ;
- **1976** : Make (simple et utile)

```
%.x : %.y
  task list
```

- Une révolution
- générique et agnostique
- très facile à prendre en main.

```
all: main.exe

lib.cma: libA.ml libB.ml
    ocamlc libA.ml libB.ml -o lib.cma

main.cmo: libB.ml main.ml
    ocamlc libB.ml main.ml -o main.cmo

main.exe: lib.cma main.cmo
    ocamlc lib.cma main.cmo -o main.exe
```

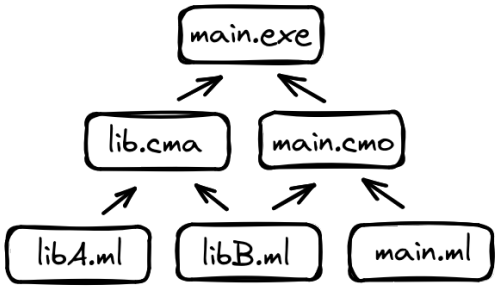

Figure 1: le treillis formé par make

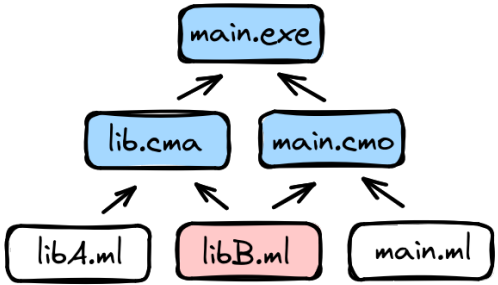

Figure 2: si l'on modifie libB.ml

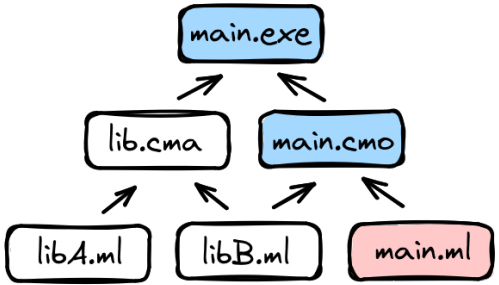

Figure 3: si l'on modifie main.ml

Les points faibles de Make

- Les dépendances ne peuvent qu'être statiques

Les points faibles de Make

- Les dépendances ne peuvent qu'être statiques
- elles doivent être répétées

Les points faibles de Make

- Les dépendances ne peuvent qu'être statiques
- elles doivent être répétées
- la stratégie de nettoyage est laborieuse

Les points faibles de Make

- Les dépendances ne peuvent qu'être statiques
- elles doivent être répétées
- la stratégie de nettoyage est laborieuse
- son côté agnostique le rend difficile pour certaines tâches spécialisés

Les points faibles de Make

- Les dépendances ne peuvent qu'être statiques
- elles doivent être répétées
- la stratégie de nettoyage est laborieuse
- son côté agnostique le rend difficile pour certaines tâches spécialisés
- logiciel ancien qui ne tient pas compte des évolutions (cloud build etc.)

Idiomes relatifs aux *Build Systems* : la minimalité

Un système de construction est minimal si il n'exécute que les tâches nécéssaires une seule fois et si il dépend transitivement des entrées ayant changées depuis la construction précédente.

Idiomes relatifs aux *Build Systems* : la minimalité

> *Un système de construction est minimal si il n'exécute que les tâches nécéssaires une seule fois et si il dépend transitivement des entrées ayant changées depuis la construction précédente.*

Pour atteindre la minimalité

- `Make` repose sur la `date de modification` d'une entrée
- et construit un graphe de dépendance de tâches basé sur **un ordre topologique**

Construction d'un système assurant la minimalité

Description des dépendances

Basé sur un Set de chaine de caractères :

```
type t

val empty : t
val of_list : filepath list -> t
val singleton : filepath -> t
include Preface.Specs.MONOID with type t := t
```

- `neutral` = `set vide`
- `concat` = `union` de deux sets.

Requêter un ensemble de dépendances

On voudrait savoir si une cible doit être modifiée selon un ensemble de dépendance. En d'autres mots:

Requêter un ensemble de dépendances

On voudrait savoir si une cible doit être modifiée selon un ensemble de dépendance. En d'autres mots:

- Si la cible n'existe pas, **on doit la créer**

Requêter un ensemble de dépendances

On voudrait savoir si une cible doit être modifiée selon un ensemble de dépendance. En d'autres mots:

- Si la cible n'existe pas, **on doit la créer**
- Si la cible existe mais:

Requêter un ensemble de dépendances

On voudrait savoir si une cible doit être modifiée selon un ensemble de dépendance. En d'autres mots:

- Si la cible n'existe pas, **on doit la créer**
- Si la cible existe mais:
  - sa date de modification inférieure à une des dates de modification des dépendances, **on doit la créer** \pause
  - sa date de modification supérieure à toutes les dates de modification des dépendances, **on ne doit pas la créer**

```
module Traverse = List.Monad.Traverse (Impure)
let get_mtimes list =
   List.map Impure.get_mtime list
```

```
module Traverse = List.Monad.Traverse (Impure)
let get_mtimes list =
   List.map Impure.get_mtime list

  (* get_mtimes : filepath list -> int Impure.t list  *)
```

```
module Traverse = List.Monad.Traverse (Impure)
let get_mtimes list =
   List.map Impure.get_mtime list
  (* get_mtimes : filepath list -> int Impure.t list  *)
  |> Traverse.sequence
  (* get_mtimes : filepath list -> int list Impure.t  *)
```

On peut maintenant décrire une fonction qui requête un set de dépendances pour savoir s'il est nécessaire de reconstruire un fichier.

On peut maintenant décrire une fonction qui requête un set de dépendances pour savoir s'il est nécessaire de reconstruire un fichier.

```
(* need_update : Deps.t -> filepath -> bool Impure.t *)
```

On peut maintenant décrire une fonction qui requête un set de dépendances pour savoir s'il est nécessaire de reconstruire un fichier.

```
(* need_update : Deps.t -> filepath -> bool Impure.t *)

let need_update deps target =
    let open Impure in
    let* exists = file_exists target in
    if not exists then return true
    else
      let* target_time = get_mtime target in
      let+ deps_times = get_mtimes (S.elements deps) in
      List.exists (fun deps_time -> deps_time > target_time) deps_times
```

On peut maintenant décrire une fonction qui requête un set de dépendances pour savoir s'il est nécessaire de reconstruire un fichier.

```
(* need_update : Deps.t -> filepath -> bool Impure.t *)

let need_update deps target =
    let open Impure in
    let* exists = file_exists target in
    if not exists then return true
    else
      let* target_time = get_mtime target in
      let+ deps_times = get_mtimes (S.elements deps) in
      List.exists (fun deps_time -> deps_time > target_time) deps_times
```

Dans les grandes lignes, c'est comme ça que `make` fonctionne.

On pourrait décrire un tâche comme étant un Applicative :

```
type 'a t = {
  deps: Deps.t
; task : 'a Impure.t }
```

```
type 'a t = { deps : Deps.t; task : 'a Impure.t }

module Applicative = Preface.Make.Applicative.Via_pure_and_apply (struct
  type nonrec 'a t = 'a t

  let pure x = { deps = Deps.empty; task = Impure.return x }
```

```
type 'a t = { deps : Deps.t; task : 'a Impure.t }

module Applicative = Preface.Make.Applicative.Via_pure_and_apply (struct
  type nonrec 'a t = 'a t

  let pure x = { deps = Deps.empty; task = Impure.return x }

  let apply fa xa = {
    deps = Deps.combine fa.deps xa.deps
  ; task = Impure.Applicative.apply fa.task xa.task }
  end)
```

```
type 'a t = { deps : Deps.t; task : 'a Impure.t }

module Applicative = Preface.Make.Applicative.Via_pure_and_apply (struct
  type nonrec 'a t = 'a t

  let pure x = { deps = Deps.empty; task = Impure.return x }

  let apply fa xa = {
    deps = Deps.combine fa.deps xa.deps
  ; task = Impure.Applicative.apply fa.task xa.task }
  end)

  let read_file filename =
    { deps = Deps.singleton filename
    ; task = Impure.read_file filename }
end
```

```
let write_file target rule =
  let open Impure in
  let* need_update = Deps.need_update rule.Task.deps target in
  if need_update then
    let* content = rule.task in
    write_file target content
  else return ()
```

On peut décrire cette règles `make` :

```
page.html: header.html content.html footer.html
  cat header.html content.html footer.html > page.html
```

```
let a_task =
  let open Task.Applicative in
  let+ header = Task.read_file "header.html"
  and+ content = Task.read_file "content.html"
  and+ footer = Task.read_file "footer.html" in
  header ^ content ^ footer
```

```
let a_task =
  let open Task.Applicative in
  let+ header = Task.read_file "header.html"
  and+ content = Task.read_file "content.html"
  and+ footer = Task.read_file "footer.html" in
  header ^ content ^ footer


{ task = ...
; deps = ["content.html"; "footer.html"; "header.html"]
} - string Task.t
```

```
let a_task =
  let open Task.Applicative in
  let+ header = Task.read_file "header.html"
  and+ content = Task.read_file "content.html"
  and+ footer = Task.read_file "footer.html" in
  header ^ content ^ footer


{ task = ...
; deps = ["content.html"; "footer.html"; "header.html"]
} - string Task.t


let page_html = write_file "page.html" a_task
```

On est même supérieur à `make` car on ne doit pas répéter les dépendances.

Par contre, comme pour `make` :

- on peut tracker que des dépendances statiques
- chaque fragment est calculé indépendament

Par contre, comme pour `make` :

- on peut tracker que des dépendances statiques
- chaque fragment est calculé indépendamment

Application partielle VS application séquentielle

```
f <$> a <*> b <*> c <*> d
a >>= f >>= g >>= h >>= i
```

Que peut-on spéculer sur ces deux lignes ?

Build System Applicatif VS Build System Monadique

On peut décire un build système monadique en remplaçant l'utilisation de `apply` par `bind` (ou `flat_map`). La différence principale réside dans :

Build System Applicatif VS Build System Monadique

On peut décire un build système monadique en remplaçant l'utilisation de `apply` par `bind` (ou `flat_map`). La différence principale réside dans :

- Un build système applicatif exécute chaque tâches indépendament et les groupes à la fin, on peut donc faire des **spéculations sur des composants statiques**, comme les dépendances. ie: `make`

Build System Applicatif VS Build System Monadique

On peut décire un build système monadique en remplaçant l'utilisation de `apply` par `bind` (ou `flat_map`). La différence principale réside dans :

- Un build système applicatif exécute chaque tâches indépendament et les groupes à la fin, on peut donc faire des **spéculations sur des composants statiques**, comme les dépendances. ie: `make`
- Un build système monadique construit une séquence dépendante entre les différentes tâches, il n y a donc pas de composants statiques par contre, ils peuvent **construire des dépendances dynamiquement** . ie: `Excell` (qui est aussi un build-system)

Build System Applicatif VS Build System Monadique

On peut décire un build système monadique en remplaçant l'utilisation de `apply` par `bind` (ou `flat_map`). La différence principale réside dans :

- Un build système applicatif exécute chaque tâches indépendament et les groupes à la fin, on peut donc faire des **spéculations sur des composants statiques**, comme les dépendances. ie: `make`
- Un build système monadique construit une séquence dépendante entre les différentes tâches, il n y a donc pas de composants statiques par contre, ils peuvent **construire des dépendances dynamiquement** . ie: `Excell` (qui est aussi un build-system)

```
#include "my_file.ml" ;;
let main () = ...
#include "termination.ml" ;;
```

`my_file.ml` et `termination.ml` sont des dépendances dynamiques, il faut lire le fichier pour les déduires.

- Nécéssaire pour construire des `build-systems` du monde réel (ou des générateurs de blog statique, pour permettre de faire des indexes de pages par exemple)
- rend l'approche de la **minimalité** (beacoup) plus complexe.

Approcher les dépendances statiques ET dynamiques

```
let dynamic_deps_example target =
   let open Dynamic in
   let* dynamic_comp = read_in_cache target <*? read_deps in
   let task =
     let+ header = Task.read_file "header.html"
     and+ content = Task.read_file "content.html"
     and+ footer = Task.read_file "footer.html" in
     header ^ content ^ footer
    in write_file_with_dynamic_comp dynamic_comp task target
```

Approcher les dépendances statiques ET dynamiques

```
let dynamic_deps_example target =
   let open Dynamic in
   let* dynamic_comp = read_in_cache target <*? read_deps in
   let task =
     let+ header = Task.read_file "header.html"
     and+ content = Task.read_file "content.html"
     and+ footer = Task.read_file "footer.html" in
     header ^ content ^ footer
    in write_file_with_dynamic_comp dynamic_comp task target
```

```
val ( <*? ) :
    ('a, 'b) Either.t Task.t
 -> ('a -> 'b) Task.t
 -> 'b Task.t
```

- `<*?` est l'opérateur `select` d'un foncteur selectif Applicatif

- `<*?` est l'opérateur `select` d'un foncteur selectif Applicatif
- En fonction de ce que renvoie la première fonction, il va exécuter la seconde où non

- `<*?` est l'opérateur `select` d'un foncteur selectif Applicatif
- En fonction de ce que renvoie la première fonction, il va exécuter la seconde où non
- Accouplé à un **cache** on peut ne pas relire inutilement un fichier.

- `<*?` est l'opérateur `select` d'un foncteur selectif Applicatif
- En fonction de ce que renvoie la première fonction, il va exécuter la seconde où non
- Accouplé à un **cache** on peut ne pas relire inutilement un fichier.

- `<*?` est l'opérateur `select` d'un foncteur selectif Applicatif
- En fonction de ce que renvoie la première fonction, il va exécuter la seconde où non
- Accouplé à un **cache** on peut ne pas relire inutilement un fichier.

```
(* val read_in_cache : filepath -> (unit, Deps.t) Either.t *)
let read_in_cache target =
    if file_is_updated file then Left ()
    else Right (get_deps_from_cache_for target)
```

- `<*?` est l'opérateur `select` d'un foncteur selectif Applicatif
- En fonction de ce que renvoie la première fonction, il va exécuter la seconde où non
- Accouplé à un **cache** on peut ne pas relire inutilement un fichier.

```
(* val read_in_cache : filepath -> (unit, Deps.t) Either.t *)
let read_in_cache target =
    if file_is_updated file then Left ()
    else Right (get_deps_from_cache_for target)
```

- `read_deps` ne sera appliquée que si la cible a été modifiée.

Foncteur Selective Applicative

- Se situe entre Applicative et Monade

Foncteur Selective Applicative

- Se situe entre Applicative et Monade
- Permet de conditionner l'exécution de la tâche suivante en fonction du résultat de la précédente (sans partager le résultat)

Foncteur Selective Applicative

- Se situe entre Applicative et Monade
- Permet de conditionner l'exécution de la tâche suivante en fonction du résultat de la précédente (sans partager le résultat)

Notions that can be expressed using an operator	apply (<*>)	select (<*?)	bind (>>=)
Arbitrary dynamic effects			✓
Conditional execution of effects		✓	✓
Speculative execution of effects		✓	
Static visibility and analysis of effects	✓	✓	
Independent effects and parallelism	✓		

Figure 4: table

Complément, Approxmation

Dans le mondes des Applicatives (et des Foncteurs), il existe un monoïde particulier (le *monoide fantôme*), `Const` pour lequel il n'existe pas d'instance de Monade mais qui offre des capacité d'analyse intéressantes :

Complément, Approxmation

Dans le mondes des Applicatives (et des Foncteurs), il existe un monoïde particulier (le *monoide fantôme*), `Const` pour lequel il n'existe pas d'instance de Monade mais qui offre des capacité d'analyse intéressantes :

```
module Const (M : Preface.Specs.MONOID) = struct
  module Applicative = Preface.Make.Applicative.Via_pure_and_apply (struct
    type 'a t = M.t

    let pure _ = M.neutral
    let apply x y = M.combine x y
  end)
end
```

En fonction de l'implémentation de `select` une **transformation naturelle** peut permettre de calculer deux types d'approximation d'exécutions de tâches :

```
module Under =
    Preface.Make.Selective.Over_applicative_via_select
      (Applicative)
      (struct
        type 'a t = M.t

        let select x _ = x
      end)
```

```
module Over =
    Preface.Make.Selective.Over_applicative_via_select
      (Applicative)
      (struct
        type 'a t = M.t

        let select = Applicative.apply
      end)
```

Par exemple en admettant un `if` et un `when` selectifs:

```
val if_ : bool Task.t -> 'a Task.t -> 'a Task.t -> 'a Task.t
val when_ : bool Task.t -> unit Task.t -> unit Task.t
```

```
let r =
  let open Const.Over in
  let x = if_ (over "a") (over "b") (over "c") in
  x *> (over "d") *> when_ (over "e") (over "f")

# over "abcdef"
```

```
let r =
  let open Const.Over in
  let x = if_ (over "a") (over "b") (over "c") in
  x *> (over "d") *> when_ (over "e") (over "f")

# over "abcdef"
```

`Over` collecte **toutes les tâches pouvant être exécutées**.

```
let r =
  let open Const.Under in
  let x = if_ (under "a") (under "b") (under "c") in
  x *> (under "d") *> when_ (under "e") (under "f")

# under "ade"
```

```
let r =
  let open Const.Under in
  let x = if_ (under "a") (under "b") (under "c") in
  x *> (under "d") *> when_ (under "e") (under "f")

# under "ade"
```

Under collecte **toutes les tâches qui vont obligatoirement se produire** (il évince les effets conditonnels)

- `Over` peut être utile, par exemple, pour installer toutes les `dépendances` possibles avant de lancer le *build* (dans le contexte, par exemple d'un *package manager*)
- `Under` donne exhaustivement les points de parallèlisme possible d'une séquence de build. Toutes les tâches `sous-approximés` peuvent être parallèlisées.

Pour conclure

Le papier *Build Systems à la Carte* présente beaucoup plus de cas d'analyses qui sont mises en pratiques dans les systèmes `dune` et `buck2`.

- Il vaut la peine d'être lu et j'en ferai surement un article de vulgarisation sur mon blog
- Il est possible de modeliser tout ce qui est présenté avec des `Arrows` (où `Choice` = `Selective`)
- Désolé pour la préparation un peu à la hâte.

Questions ou remarques ?