

The elevator problem

*A brief contextualized introduction to **finite state machines**
in a **type-safe** context*

Xavier Van de Woestyne
xaviervdw@gmail.com - *margo.com*

BreizhCamp 9 - 2019

Bonjour !

- Belge, vivant à Lille, travaillant à Paris ;
- *Data Engineer* chez **Margo Bank** ;
- J'aime bien programmer
(OCaml, F#, Haskell, Erlang/Elixir, Kotlin, Io, Elm) ;
- **Phutur** : "*Useless software with useful language*" ;
- **LilleFP** : on recherche toujours des *speakers* !

Objectifs de la présentation

- Raisonner la notion de **programme à états** ;
- râler sur certains ascenseurs ;
- fantasmer sur l'implémentation d'ascenseurs dans des langages indaptés (pour ce *use-case*).
- présenter **les machines à états** (finis) ;
- en implémenter dans le contexte d'un langage **statiquement typé** ;
- proposer un *crash-course* expresse sur le typage.

Caveat emptor !

- Une présentation **subjective** et un peu idéologique ;
- les langages utilisés sont probablement (surement) parfaitement inadaptés à ce genre d'exercice en situation réelle ;
- pas du tout de couverture sur la notion d'effet (le temps est mon ennemi) ;
- je met toujours trop de texte sur mes diapositives¹

¹Mais je n'ai toujours pas trouvé comment mettre des speakers notes avec Beamer.

Mise en contexte

Il était une fois, dans un espace de coworking à Paris ...
un seul ascenseur idiot pouvant accueillir
~6 personnes pour 8 étages

Le tout, couplé à de très amusants aléas

1. Un seul bouton d'appel

- Donc pas d'optimisation des trajets

2. Empilement "brute" dans la pile d'appel

- J'ai réussi à aller jusqu'à 13 !

Un problème de *design* complexe

Facilement transposable dans nos problématiques de tous les jours

Des états et de l'arbitrage

1. Arbitrage

- Choix de stratégies (attente longue VS trajet long) ;
- quand peut-on autoriser la réouverture des portes ;
- intersection entre la prise de décision simple et automatique (comment se comporter vis à vis du poids maximum supporté).

2. L'implémentation

- Une modélisation plus complexe qu'il n'y paraît ;
- Chaque **état** de l'ascenseur possède son propre domaine : on aimerait **rendre les états impossibles, impossibles** ;
- Comment *scaler* d'un ascenseur à un cluster d'ascenseurs ?

Un programme à états

- L'ascenseur est en attente ;
- l'ascenseur est en cours de trajet ;
- quand est-il possible d'appeler l'ascenseur ;
- quand est-il possible d'ouvrir les portes ;
- etc.

Les machines à états

on entre dans le vif du sujet !

Les machines à états

Elles définissent un ensemble **d'états** et de **transitions** possibles entre les états.

Elles peuplent le monde de l'informatique :

- Pour la définition de protocoles (Sockets, TCP, RAFT etc.) ;
- pour la structure de programme (Elm-Architecture/Redux).

Malgré leur popularité, j'ai² vu peu de développeurs s'en servir pour structurer les états de leurs programmes.

²Du haut de ma toute petite expérience.

Première approche : les états implicites

- Ne pas être explicite sur les états du programme ;
- on définit ces états par des environnements (variables mutables etc.)
- ça rend le programme dur à raisonner (sur l'intégrité des transitions par exemple) ;
- ça impose trop souvent des **assertions à l'exécution**.

Utilisation de machine à états concrètes

- Modélisation du programme comme une machine *abstraite* ;
- propose un ensemble d'états (ici, fini) ;
- ne peut être que dans un seul état à la fois ;
- des événements peuvent déclencher une transition d'état ;
- pour chaque état il existe une suite légale de transition ;
- ces transitions sont exprimées comme une association d'événements à d'autres états.

Utilisation de machine à états concrètes

Erlang décrit les machines à états finies comme un ensemble de relation de cette forme :

$$State(S) * Event(E) \rightarrow Action(A), State(S')$$

Si on est dans l'état **S** et que l'événement **E** se produit, alors on peut exécuter l'action **A** et faire une transition vers l'état **S'**.

Par exemple

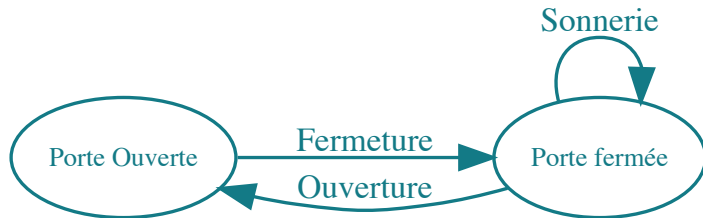


Figure 1: Une porte représentée comme une machine à états finis

Supports natifs

1. Dans des langages :
 - **Erlang** et **Elixir** via `gen_fsm` ;
 - **SCADE** (de Esterel) ;
 - **Mbeddr C** ;
 - **Pure Data** (et autres MSP Like)
2. Comme outil de description
 - Grafcet
 - SMC
 - CHSM
3. Une nouvelle vie dans les jeux-vidéo

Pro/Cons des machines à états explicites

1. Avantages

- Elles décrivent formellement le cycle de vie d'une application ;
- ça apporte de la documentation (pour d'autres développeurs ou pour le "métier") ;
- l'ensemble des états et des transitions sont facilement testables ;
- ça offre un outil de raisonnement accessible.

2. Inconvénients potentiels

- Peu imposer du *boilerplate* à la définition ;
- dur à implémenter dans certains langages (récursion terminale VS *open recursion*).

Cas pratique : implémentation d'une machine à états finie dans un langage statiquement typé

Un programme très simple

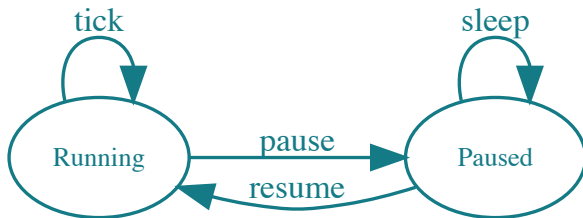


Figure 2: Une machine "discutablement utile".

Utilisation de OCaml

- Langage issu de la recherche française ;
- qui, malgré ce que l'on entend, possède des utilisateurs industriels ;
- un langage expressif et multi-paradigme ;
- très facile à prendre en main ;
- qui fait office d'inspiration pour beaucoup d'autres langages ;
- avec un système de types riche et expressif.

Objectifs

- Construire une machine à états très simple ;
- découvrir progressivement des fonctionnalités liés aux systèmes de types ;
- être aidé au maximum par le compilateur pour rendre des états impossibles ... impossibles ;
- comprendre comment les systèmes de types algébriques nous permettent de modéliser des systèmes ;
- faire une promotion non maquillé du langage OCaml.

Le système de types de OCaml : Alias et Produits

Kotlin :

```
typealias Firstname = String
typealias Lastname = String
typealias Point = Tuple<Int, Int>
```

```
data class Named(val firstname: Firstname, val name: Lastname)
```

OCaml :

```
type firstname = string
type lastname = string
type point = (int * int)
```

```
type named = {firstname: firstname; name: lastname}
```

Le système de types de OCaml : Sommes

```
type gender =  
  | Male  
  | Female  
  | Other of string
```

```
let m = Female
```

```
val m : gender = Female
```

Le système de types de OCaml : Sommes

```
type gender =  
  | Male  
  | Female  
  | Other of string
```

```
let to_string gender = match gender with  
  | Male -> "male"  
  | Female -> "female"
```

Warning 8: this pattern-matching is not exhaustive.

Here is an example of a case that is not matched: `Other _`

```
val to_string : gender -> string = <fun>
```


Le système de types de OCaml : Sommes

Kotlin :

```
sealed class Option<out T: Any>
data class Some<out T: Any>(val value : T) : Option<T>()
object None : Option<Nothing>()
```

OCaml :

```
type 'a option =
| Some of 'a
| None
```

Le système de types de OCaml : Sommes

Kotlin :

```
sealed class Option<out T: Any> {  
    abstract fun <A: Any> map(f: (T) -> A) : Option<A>  
}  
  
data class Some<out T: Any>(val value: T) : Option<T>() {  
    override fun <A: Any> map(f: (T) -> A) : Option<A> = Some(f(value))  
}  
  
object None : Option<Nothing>() {  
    override fun <A: Any> map(f: (T) -> A) : Option<A> = None  
}
```

OCaml :

```
type 'a option = Some of 'a | None  
let map f opt = match opt with Some x -> Some(f x) | None -> None
```

Séparation des définitions et des consommations

```
sealed class Option<out T: Any>
data class Some<out T: Any>(val value: T) : Option<T>()
object None : Option<Nothing>()

// Utilisation de smart-cast
fun <T, A> map(opt: Option<T>, f: (T) -> A) : Option<A> =
    when(opt) {
        is Some -> Some(f(opt.value))
        is None -> None
    }
```

Cette approche peut tout de même construire des soucis au niveau de la variance.

Les types algébriques

- Les sommes et les produits sont des **types algébriques**
(parce que leur domaine est égal à la somme ou la multiplication des domaines de leurs membres)
- Couplés avec de la **correspondance de motifs**, et à la récursion, ils permettent de décrire toutes forme de structures.

Note sur le langage de module de OCaml

- Une unité de compilation qui sépare la signature de l'implémentation ;
- permet de gérer la visibilité des types/fonctions ;
- permet d'abstraire certains types ;
- permet beaucoup d'autres choses ...

```
module My_module : sig
  type name = string
  val hello : name -> unit
end = struct
  type name = string
  let hello = Format.printf "Hello %s\n"
end
```

Retour à notre machine à états

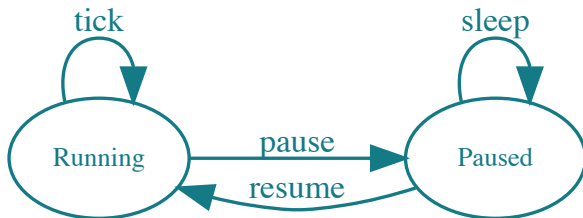


Figure 3: Retour à notre cas d'usage !.

Une première approche

```
type time = int
type step = Running of time | Paused of time

let start () = Running 0
```

Et un premier problème :

```
type time = int
type step = Running of time | Paused of time

let start () = Running 0
let resume state = match state with
  | Paused x -> Resume x

let pause state = match state with
  | Running x -> Paused x
```

Correspondance de motifs non exhaustive et `pause (Paused 10)` provoque une erreur à l'exécution !

Une autre approche

```
type time = int
type running = Running of time
type paused = Paused of time

let start () = Running 0
let resume (Paused time) = (Running time)
let pause (Running time) = (Paused time)
```


Seems better

```
start ()
```

```
|> pause
```

```
|> resume
```

```
|> pause
```

```
let tick (Running x) = Running (x + 1)
```

```
let sleep time (Paused x) = Paused (x + time)
```

```
start ()  
|> tick  
|> pause  
|> sleep 10  
|> resume  
(** Ok ! *)
```

```
start () |> resume
```

Error: This expression has type `paused * 'a -> running * 'a` but an expression was expected of type `running * time -> 'b` Type `paused` is not compatible with type `running`

Ce qui nous donne comme interface générale :

```
type time = int
type running = Running of time
type paused = Paused of time

val start : unit -> running = <fun>
val resume : paused -> running = <fun>
val pause : running -> paused = <fun>
val tick : running -> running = <fun>
val sleep : time -> paused -> paused = <fun>
```

Comment gérer les fonctions communes entre les états ?

```
val le_temps_passe : time -> ??? -> ???
```

```
val le_temps_passe_paused : time -> paused -> paused
```

```
val le_temps_passe_running : time -> running -> running
```

Ce n'est pas très agréable ...

En séparant le temps de l'état

```
type time = int
type running = Running
type paused = Paused
type 'a state = ('a * time)

val start : unit -> running state = <fun>
val resume : paused state -> running state = <fun>
val pause : running state -> paused state = <fun>
val tick : running state -> running state = <fun>
val sleep : time -> paused state -> paused state = <fun>

val le_temps_passe : time -> 'a state -> 'a state
```

Almost done !

- Bien que très verbeux, et long, on s'approche de l'objectif
- par contre : `le_temps_passe ("hello", 20)` est un programme valide...

Les variants polymorphes

- Des types sommes qui ne doivent pas être déclarés préalablement
- qui introduisent une notion de variance :

```
let f x = match x with  
  | `Foo -> ("foo", 0)  
  | `Bar x -> ("bar", x)
```

```
val f : [< `Bar of int | `Foo ] -> string * int = <fun>
```

```
let f' x = match x with  
  | `Foo -> ("foo", 0)  
  | `Bar x -> ("bar", x)  
  | _ -> ("hmmm", 1)
```

```
val f' : [> `Bar of int | `Foo ] -> string * int = <fun>
```

Utilisation de types fantômes

```
module My_fsm : sig
  type time = int
  type 'a state
end = struct
  type time = int
  type 'a state = time
end
```

On abstrait le type **state**


```
module My_fsm : sig
  type time = int
  type 'a state

  val start : unit -> [`Running] state
  val resume : [`Paused] state -> [`Running] state
  val pause : [`Running] state -> [`Paused] state
  val le_temps_passe : int -> 'a state -> 'a state
  val to_time : 'a state -> time
end
```

```
module My_fsm = struct
  struct
    type time = int
    type 'a state = time

    let start () = 0
    let resume x = x
    let pause x = x
    let le_temps_passe time  x = x + time
    let to_time x = x
  end
end
```

Avec une approche plus uniforme

```
val le_temps_passe : int -> [< (`Running | `Paused) as 'a ] state -> 'a state

module My_fsm = struct
  struct
    type time = int
    type 'a state = Running of time | Paused of time

    let start () = Running 0
    let resume x = match x with Paused t -> Running t
    let pause x = match x with Running t -> Paused t
    let le_temps_passe time x = match x with
      | Running x -> Running (x + time)
      | Paused x -> Paused (x + time)
  end
end
```

resume et pause ne sont pas exhaustive ...

```
let resume x =  
  match x with  
  | Paused t -> Running t  
  | _ -> assert false
```

```
let pause x =  
  match x with  
  | Running t -> Paused t  
  | _ -> assert false
```

Astuce pour la correspondance de motif exhaustive... c'est un peu triste

Types algébriques généralisés (GADTs)

Les **GADTs** offrent, entre autres, la possibilité d'indexer chaque constructeurs d'une somme avec un type :

```
type _ example =  
  | Foo : int example  
  | Bar : float example  
  
let x = Bar  
val x : float example = Bar
```

Ils permettent de rendre exhaustif des correspondances de motifs

```
let f x = match x with  
  | Foo -> "foo"
```

```
val f : int example -> string = <fun>
```

Au contraire des types fantômes :

- Ne nécessite pas d'abstraire le type sur lequel on travail ;
- permet de construire des fonction non-surjectives ;
- garder des correspondances de motifs exhaustives "sur des fragments de domaines".

Implémentation avec des GADTs

```
type time = int
type _ state =
  | Running : time -> [`Running] state
  | Paused : time -> [`Paused] state

let start () = Running 0

let resume (Paused x) = Running x
let pause (Running x) = Paused x

let tick (Running x) = Running (x + 1)
let sleep time (Paused x) = Paused (time + x)
```

```
start ()  
  |> tick  
  |> pause  
  |> resume  
  |> tick  
[ `Running ] state = Running 2
```



```
start ()  
  |> tick  
  |> pause  
  |> sleep 10  
  |> resume  
  |> tick
```

```
[ `Running ] state = Running 12
```

```
start ()  
  |> sleep 10
```

Error: This expression has type [`Paused] state -> [`Paused] state but an expression was expected of type [`Running] state -> 'a These two variant types have no intersection

Objectif réussi !

Les GADTs pour contraindre des états

- Ils permettent d'éviter les motifs rémanent ;
- ils offrent des outils d'égalité de type fins ;
- tout en préservant une manière *idiomatique* de programmer ;
- malheureusement présents dans peu de langages.
- Avec les types rékursifs on peut composer différentes machines à a états pour construire des scénarios plus complexes.

Aller plus loin

- Rendre de plus en plus de partie de son programme "sures" (par exemple, la distinction entre les listes/vides non vides, option, results) ;
- Intégrer la notion d'effets dans les machines à états ;
- construire des machines à états génériques à coup de monades indexées libres (pour représenter un triplet de Hoare) ;
- intégrer des types dépendants pour la structure de son programme ³

³Pendant longtemps, les types dépendants étaient réservés à l'élaboration de preuves, depuis l'apparition de langages hybrides comme **Idris**, il est possible de choisir "à la demande" quand être total ou ne pas l'être !

Conclusion

- Les machines à états décrivent des problèmes récurrents ;
- elles permettent la séparation systématique entre les états et les actions ;
- dans un langage avec un système de type riche elles amènent certaines garanties ;
- elles s'adaptent à plusieurs problèmes, le jeu vidéo, le web, les systèmes distribués, l'embarqué ;
- Kotlin est quand même plus verbeux que OCaml...

Elles peuvent amener une question d'arbitrage entre **le coût de mise en place** et **le coût d'usage**. Dans un langage ML, ce serait dommage de s'en priver.

Fin

Merci ! (si vous avez envie de venir présenter quelque chose
à LilleFP, n'hésitez pas à m'en parler)