The elevator problem

*A brief contextualized introduction to **finite state machines** and some **data structures***

Xavier Van de Woestyne

Touraine Tech 2019

Hello !

- Xavier Van de Woestyne (Bruxelles, Lille, Paris) ;
- *Data Engineer* chez **margo.com**
- J'aime bien programmer (OCaml, F#, Haskell, Erlang/Elixir, Kotlin, Io, Elm) ;
- https://xvw.github.io, **xvw** sur Gitlab et Github ;
- **twitter.com/vdwxv** et **xvw@merveilles.town** ;
- **Phutur** : *useless software with useful languages* ;
- **LilleFP** : *Meetup* approximativement bimestriel ;

- j'ai une "technique" de préparation de présentation assez particulière :
  - j'essaie de découvrir quelque chose ;
  - j'essaie de faire la promotion de quelque chose ;
  - **je fantasme sur une "*perle*"** ;
  - **je râle sur quelque chose.**

Objectifs de la présentation

- Raisonner la notion de **programme à états** (*stateful*) ;
- présenter un exemple **concret** de programme à états (les ascenseurs) ;
- proposer des perspectives d'implémentation **commodes** ;
- voir les **forces** et les **faiblesses** des machines à états par opposition aux "objets" classiques ;
- raisonner (un peu) **l'expérience utilisateur** ;
- utilisation de langages moins *mainstreams* (et statiquement typés) ;
- fantasmer l'implémentation "potentielle" d'ascenseurs... (en utilisant des langages inadaptés à l'implémentation réelle d'un ascenseur)

Hors périmètre

- Implémentation "réelle" de l'ascenseur ;
- focus sur l'implémentation logique.

Caveat emptor !

- L'objectif de la présentation est de tâcher d'être **pédagogique** ;
- c'est une présentation très **idéologique** (et subjective) ;
- le **raisonnement** sera donc privilégié à la présentation de code ;
- ce serait sûrement très bête d'implémenter un ascenseur de cette manière…
- 40 minutes… c'est très court ^^' (+10 de questions réponses).

Mise en contexte

Une vraie histoire !

Dans un espace de co-working *parisien, il y a* ***un*** *ascenseur ...*

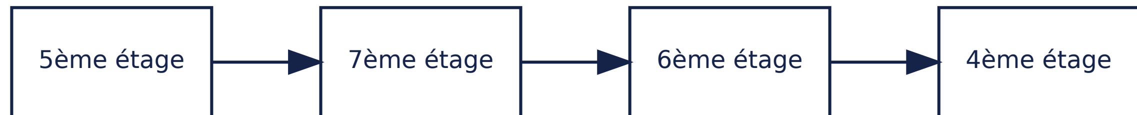
- Un ascenseur pour **8 étages** ;
- pouvant accueillir ∼6 personnes ;
- avec **un seul** bouton d'appel ;
- tombant souvent en **panne**.

Mise en contexte : **un seul** bouton d'appel

Pouvant rendre le trajet **très frustrant**. Par exemple :

- si l'ascenseur est à l'étage 5, et qu'il se rend à l'étage 7 ;
- que j'appelle l'ascenseur à l'étage 6, pour descendre à l'étage 4 ;
- l'ascenseur s'arrêtera à l'étage 6, puis continuera vers l'étage 7 ;
- a cause d'un soucis, l'ascenseur s'arrêtera **toujours** à l'étage 6 ;
- pour ensuite descendre à l'étage 4 ...

Mise en contexte : **un seul** bouton d'appel



Optimisation du trajet

L'absence de deux boutons (monter/descendre) rend l'optimisation du trajet impossible.

Autre aléa (assez amusant)

Avant la maintenance N dûe à une Nème panne

- Si j'appuie **N** fois sur le bouton d'appel, en attendant l'ascenseur ;
- arrivé à mon étage, il ouvrira/fermera ses portes **N** fois...
- a l'époque je m'étais arrêté à 13.

Autre aléa (assez amusant)

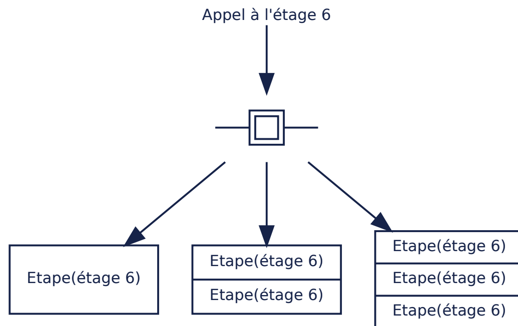


Figure 1: Stack ftw

Voici le "pourquoi" de cette présentation

A chaque fois que j'ai dû attendre cet ascenseur... j'y pensais.

Un problèmes sur plusieurs fronts

L'arbitrage

- Décider quelles stratégies adopter (attente longue ou trajet long ?) ;
- quand "rouvrir" la porte ?
- Intersection intelligente entre "la prise de décision automatique" et le "raisonnement simple". (Par exemple, comment se comporter vis à vis du poids maximum supporté par un ascenseur ?)

L'implémentation

- Modèlisation plus complexe qu'il n'y parait ;
- choix de structures de données adéquates ;
- **rendre les états impossibles... impossibles** ;
- **scalabilité** : comment augmenter le nombre d'ascenseurs ?

Raisonnement sur un programme "à états"

- Un programme qui "*se souvient*" des événements précédents ;
- et qui peut effectuer un transition vers un autre état en se basant sur son état courant.

Une collection d'ascenseurs est régie par une collection de règles et d'états

- Que se passe-t-il quand on appelle un ascenseur ?
- Peut-on ouvrir des portes qui sont déjà ouvertes ?
- Peut-on fermer des portes qui sont déjà fermées ?
- A quel étage se trouvent les ascenseurs ?

Modèlisation naïve : les états implicites

- Ne pas être explicite sur les états ;
- qui deviennent définis par des environnements (des variables mutables) ;
- ce qui rend le programme **dur à raisonner**, notamment sur **l'intégrité** des transitions ;
- impose (trop) souvent des **assertions à l'exécution**.

On devrait rendre les états explicites !

- Définitions "formelles de états" ;
- rendre les transitions **explicites** ;
- rendre les états impossibles... impossible.

Machines à états finis

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

Machines à états finis

Erlang décrit les machines à états finis comme un ensemble de relations de cette forme :

$$State(S) * Event(E) -> Action(A), State(S')$$

telle que

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

Machines à états finis (par exemple)

Figure 2: Une porte représentée comme une machine à états finis

Autres exemples

- Certains fragments de manipulation d'expressions régulières ;
- la **Elm-Architecture** (ou **Redux**) ;

Support natif dans certains langages

Langages

- **Erlang** et **Elixir** (via `gen_fsm`)
- **SCADE** (de Esterel)
- **Mbeddr C**
- **Pure Data** (et autres MSP Like)

Descriptions de FSM

Grafcet, SMC, CHSM

Nouvelle vie avec les éditeurs de jeux vidéo.

Pro/cons des machines à états finis

Avantages

- Elles décrivent **formellement** le cycle de vie d'applications :
  - utile pour la documentation (et pour les autres développeurs) ;
  - ça facilite la collaboration avec le "métier" ;
- L'ensemble des états et des transitions sont facilement testables (par exemple, avec du **PBT**).

Inconvénients

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

Implémentation dans un langage avec un système de types riche

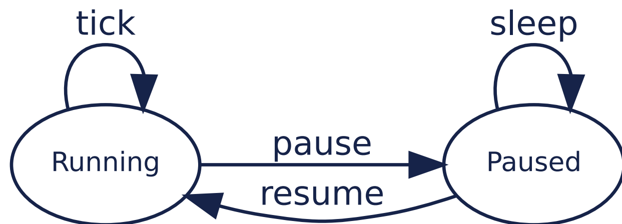


Figure 3: très proche des portes

Avec interdiction d'utiliser `tick` quand on est en pause, et `sleep` quand on est en court.

Utilisation de OCaml

- Langage issu de la recherche française ;
- beaucoup d'utilisateurs industriels (et de *success stories*) ;
- un langage très expressif (multi-paradigme) ;
- facile à prendre en main ;
- inspiration de beaucoup de *nouveaux langages* ;
- avec un système de types algébriques.

Types produits

```
(int, float) {x : int; y: int, name: string}
```

Types sommes

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

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

Types exponentiels

```
(int -> int -> int) (float -> int) ('a -> 'a)
```

```ocaml
let hello = function
  | Male -> "Bonjour monsieur"
  | Female -> "Bonjour madame"
```

Warning 8: this pattern-matching is not exhaustive. Here is an example of a case that is not matched: `Other _`

```
let hello = function
  | Male -> "Bonjour monsieur"
  | Female -> "Bonjour madame"
  | Other s -> Bonjour " ^ s
```

Warning 8: this pattern-matching is not exhaustive. Here is an example of a case that is not matched: `Other _`

Variants polymorphes et types algébriques généralisés (GADT)

```ocaml
(* Variants polymorphes *)
`Foo; `Bar of int; [`Foo | `Bar of int]

(* GADT *)
type _ t =
  | A : int t
  | B : float t
  | C : 'a -> 'a t

let f = function
  | A -> "valeur A"
```

Implémentation de la machine

```
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 !

Render impossibles les états impossibles !

Types récursifs et composition

Les types en OCaml peuvent être récursifs :

```
type 'a list =
  | []
  | ( :: ) ('a, 'a list)
```

On peut donc **composer** des fragments de machines à états et construire des scénarios bien plus complexes. (Par exemple, des états d'ascenseurs).

Structures algébriques et structures de données relatives à l'implémentation

Persister un état dans un style fonctionnel

Il est nécéssaire de persister l'état des ascenseurs dans la machine :

- Relayer l'état de chaque ascenseurs dans l'état du *cluster* ;
- utilisation d'une **monade d'état** (à la Haskell).

Collection et piles

Par exemple, pour structurer la liste des étapes de l'ascenseur (étage 1 -> 2 etc.)

- Liste chainée ;
- Zipper.

Heuristiques

Comment définir quel ascenseur, plutôt qu'un autre, doit venir. C'est un sujet de recherche encore assez actif, cependant, comme les ascenseurs ne se déplacent pas encore sur deux axes on peut facilement poser les variables pour définir un **score** :

- Le nombre d'étapes nécéssaire pour se rendre au lieu d'appel ;
- la probabilité d'interruption.

Aller plus loin, extension du système

Actuellement, l'ajout d'étape et de transition implique la modification de la bibliothèque initiale : **The expresion problem** :

The expression problem is a new name for an old problem.
The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts).

Propositions

- GADT + `open types`, mais perte de sûreté ;
- approche **Tagless final**.

Et dans la vraie vie ?

(où un langage fonctionnel serait adapté :))

- Les machines à états finis décrivent des problèmes récurrents ;
- elles permettent une **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 ;

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.

Aller plus loin avec les types

- Composer avec les combinateurs fonctionnels usuels (implique l'utilisation d'une **monade indexée**) ;
- des machines à états basées sur des **types dépendants** (Idris).

Conclusion sur les systèmes de types

Fin

Merci ! (On recherche toujours des speakers pour **LilleFP**)