Rapport de stage

T.Renaux Verdiere

2023–2024

Résumé

fjsdqlkfdq

Table des matières

I Objectif du Stage

Ce stage a pour objectif, le développement d'une bibliothèque traitant d'automates à partir d'expression rationnelle en *Haskell*. De plus, cette bibliothèque doit être accompagnée d'une description formelle. Que ce soit des algorithmes et même des structures utilisées. Il a de plus été envisagé assez tôt dans ce stage, la mise en place d'une interface graphique qui pourrait améliorez l'accessibilité de cette bibliothèque.

La bibliothèque d'automate ne devait pas seulement permettre de créer un automate, elle devait aussi mettre en place des fonctions usuelles sur les automates. Toutes ces fonctions sont détaillées dans la spécification technique de la bibliothèque. De même, le développement d'un parser de chaine de caractère vers une expression régulière et d'un fichier *JSON* vers un automate devait être mis en place.

Pour améliorer la lisibilité, une représentation graphique à l'aide du langage *dot* et de ces utilitaires devait être mis en place. Enfin, un site web avait était prévue pour mettre en place une interface graphique améliorant l'accessibilité de toutes les fonctionnalités citées précédemment.

1 Organisation du Travail

Le développement de cette bibliothèque se fait parallèlement en *Haskell* et en *Rust*. En effet, nous sommes un binôme sur ce projet. Bien que les objectifs soient les mêmes, les implémentations n'ont rien à voir. La différence entre les deux langages est si grande que très peu de ressemblance peuvent être trouvé.

L'organisation du travail pour ce stage, a été mis en place lors de notre première réunion qui s'est tenu le premier jour de ce stage. Il a alors été convenu de mettre en place une réunion hebdomadaire. De ce fait, toutes les semaines, une réunion pour parler des avancées, difficultés et objectif ont été mises en place. De manière générale, de telle réunion avait un temps variable pouvant aller de 20 minute à pour les plus longs plus d'une heure.

J'ai pour ma part après cette première réunion qui nous as donnés les objectifs de ce stage, mis en place un diagramme de *Gantt*, celui de la figure 1. Comme on peut le voir sur cette figure, chaque couleur corresponde à une tâche en particulier. Ce diagramme aillant été fait au début de mon stage, certain objectif ont pu prendre plus de temps que prévue. La correspondance couleur, objectifs est celle ci-dessous :

- Rouge : Mise en place d'un type d'**expression régulière**, ainsi que d'**automates**. Puis implémenter la conversion d'expression vers automate à l'aide de l'algorithme de *Glushkov*.

- Bleu : Développement d'un **Lexer**, aillant pour but de convertir une chaine de caractère en **expression régulière**.
- Vert : Développement de diverses fonctions sur des automates. Que ce soit des *'getters'/ 'setters'* ou encore des tests de propriétés comme l'homogénéisée ou encore savoir s'il est standard (se référer au document technique).
- Jaune : Création d'une conversion d'un automate en *Dot*. Pour une représentation graphique qui as pour d'être d'être, elle aussi, développer.
- Violet : Mise en place du site web qui devait permettre d'obtenir une représentation graphique et une meilleure accessibilité.
- Orange : Enfin, ces deux dernières semaines ont été laissées pour peaufiner chacune des parties précédentes, mais aussi de rédiger ce rapport de stage.

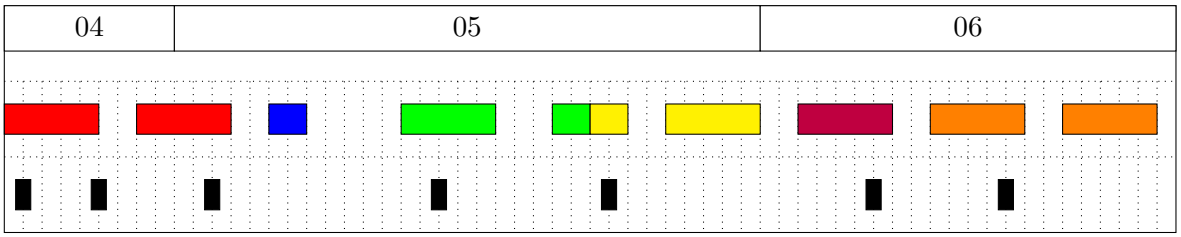

FIGURE 1 – Diagramme de *Gantt* mise en place pour le stage. Les rectangles noirs correspondent au jour de réunion.

2 Difficultés rencontrées

J'ai pu au cours de ce stage, rencontrées de nombreuses difficultés. Pour les exposées, j'ai décidé de diviser en deux catégories. Les difficultés liées au développement, à la partie théorique et les problématiques spécifique d'un stage.

2.1 Problème lié au stage

Ce stage constitue ma première expérience proche d'un travail en entreprise. De ce fait, j'ai pu rencontrer les problèmes que je pense nombreux ont. Notamment pendent quelque jour, notamment la première semaine, nous étions en télétravail. De nombreux problèmes survienne par ce mode de travail. Tous d'abord la façon de gérer ces horaires. Mr Mignot nous as laissé gérer nos horaires nous même ce qui à fait en sorte que pendant le télétravail j'ai eu du mal à ne pas trop travailler et même dans l'autre sens parfois. Ensuite, on a les problèmes du quotidien comme des pertes de connexions qui rendent certaine journée pratiquement impossible à avancée. Enfin, je n'ai pas eu de chance lors de ces derniers mois et je suis tombé malade, ce qui a accentué encore plus ce télétravail.

Un autre problème et survenue, celui-ci concernait le matériel. En effet, mon ordinateur portable n'avait selon mon système d'exploitation plus de mémoire vive. J'ai du alors réinstaller un nouvel os. Cependant, je n'avais pas prévue qu'il existait encore des os qui ne se lancé pas sur *systemd*. Sans rentrer dans les détailles, ce problème ma pris énormément de temps à régler. C'est un problème, car le gestionnaire de paquet (s'il peut l'appeler comme ça) *Nix* (qui est l'outil le plus utilisé pour la création de site en *Haskell*) doit ce lancé à l'aide de *systemd*. Sait d'ailleurs une des raisons aillant poussé vers la migration du développement d'un site web vers une application *Gtk*. Se référer à la sous-section suivante et à la partie résultat de ce compte rendu.

2.2 Difficultés de développement du projet

Avant tout, j'ai n'ai commencé à apprendre ce langage que depuis décembre. Dès lors que j'ai obtenu ce stage et que j'ai appris que le sujet nécessite le *Haskell*. J'ai donc demandé des ressources à Mr Mignot et j'ai suivi ces documentations. Notamment la lecture des livres *Get Programming with Haskell* et *Haskell in Depth* [1, 2]. Pour combler celà, j'ai aussi suivie certain cours vidéo disponible sur *Youtube* [3, 4]. De ce fait, malgré avoir commencé assez tôt l'apprentissage de ce langage, comme évoquer dans de nombreuse ressource la courbe d'apprentissage du *Haskell* est exponentielle. Donc de nombreux problèmes que je vais évoquer ci-dessous aurait pu ne pas avoir vue le jour, si j'avais une plus grande expérience avec ce langage.

Le premier problème a été rencontré lors de la mise du convertisseur de chaine de caractère vers expression. Les modules utilisés ont été comme conseille par Mr Mignot, *Alex* et *Happy*. Le problème a été la documentation de ces modules. Cette manière de documentation basée uniquement sur ce que font chaque fonction et non de comment on peut les utiliser et la non-présence d'exemple a rendu ce développement bien plus long. Ce même problème a été vu lors de l'utilisation des modules *GraphViz* qui permet la représentation d'un graph. La documentation est totalement horrible. De plus, le module étant très peu utilisé il n'existe pratiquement aucun exemple sur internet. J'ai donc du programmer a taton et me référer au code source du module pour comprendre son mode de fonctionnement.

Le deuxième problème, j'ai l'ai eu bien plus tard dans le développement. Cependant celui-ci ma fait perdre pratiquement deux voir trois semaines. Pour la mise en place site web, il avait été convenu lors d'une réunion, d'utilisé le paradigme de la *Programmation fonctionnelle réactive*.

Enfin la dernière difficulté rencontrée à été la rédaction du rapport formelle dans un premier temps. Bien que j'aie entrepris son écriture en même temps que le développement de la bibliothèque. Le formalisme nécessaire et les définitions qu'il faut introduire pour que le document se suffise à lui-même a rendu cette tâche très chronophage. De plus, n'aillant jamais réellement été confronté a cet exercice lors de la licence, il a été

dur de s'y adapté. Je me suis d'ailleurs rendu compte du nombre de relectures nécessaire pour un tel document. Je ne pensais pas qu'il fallait en faire au temps. De même pour la bibliographie qui doit si elle est citée définir les concepts de la même manière que ceux du document.

Cette difficulté de rédaction s'est aussi vue sur la fin de ce stage, lors de la rédaction de ce rapport. Aillant maintenant l'habitude des rapports de projet avec tous ceux rédiger lors de la licence, je ne pensais pas que celui-ci aller être si dure. La différence entre ce rapport et ceux précédent est surtout la différence de contenu. L'objectif de celui-ci ne vise pas à montrer toutes les facettes du projet sur laquelle j'ai travaillé, mais plutôt la manière dont j'ai travaillé déçu est ce que j'ai appris.

II *Haskell*

L'*Haskell* est un langage extrêmement différent de ceux vue au cours de la licence. Tout d'abord, il s'agit d'un **langage fonctionnel**. Bien que nous ayons vue ce paradigme par le biais du langage *OCaml*, nous navons jamais été aussi loin. Dans le développement notamment, nous n'avons par exemple jamais mis en place de parser ou bien même d'application graphique. De plus, le concept de **programmation pure** est poussé à un extrême dans ce langage. Il existe une séparation hermétique entre ce qui est **pure** et ce qu'il ne l'ait pas.

1 Programmation pure

> *'A pure function is a function that, given the same input, will always return the same output and does not have any observable side effect.'* [5]

Pour revenir sur le concept de **programmation pure**, dans un langage fonctionnelle, on parle alors de **fonction pure** (Le paradigme, ne permettant que la création de fonction). On définit une fonction présente dans un programme comme étant **pure** si et seulement si :

- La fonction est mathématiquement **déterministe**. Par cela, nous entendons que pour une fonction f, il ne peut y avoir $f(x) = y_1$ et $f(x) = y_2$ avec $y_1 \neq y_2$. En programmation, on conçoit que de nombreuse fonction sont déterministes tels que le calcule du n-ième terme de la suite de Fibonacci. On peut voir sur la figure 2, une implémentation de cette fonction (L'implémentation présentée ici, est la version intuitive. Il existe cependant des méthodes bien plus performantes pour se calcule, se référer à l'article de blog [6]). Il vient de manière logique que notre fonction est déterministe

```
fibo :: Int -> Int
fibo 0 = 0
fibo 1 = 1
fibo n = fibo' 0 1 n
  where
    fibo' _ m2 1  = m2
    fibo' m1 m2 m = fibo' m2 (m1 + m2) $ m - 1
```

FIGURE 2 – Code *Haskell*, du calcul du n-ième terme de Fibonacci.

- La fonction doit ne doit pas produire d'**effet de bord**. Un **'side effect'** comme nommé en anglais, est une qualification d'une action qui modifie ou dépend de son environnement extérieur lors d'un calcule. Un exemple très simple peut être trouvé dans l'affichage d'un nombre sur la sortie standard. Dans le langage **C**, cet affichage correspondrait au code de la figure 3. Ces **effets de bord** sont extrêmement problématique, car ils peuvent causer des erreurs non prises en

compte par celui qui a conçu la fonction et même par la personne l'utilisant. Dans le cadre d'un affichage, ces erreurs ne sont pas forcément problématiques, mais dans le cas de modification de variable global par exemple, ou même du contenu d'un fichier cela pourrait compromettre l'intégrité du code.

```c
#include <stdio.h>

void print(int n) {
    printf("%d", n);
}
```

FIGURE 3 – Code *C*, d'un affichage sur la sortie standard.

2 Les spécificités du langage

Comme évoquée plus tôt, l'*Haskell* met en place une séparation hermétique entre les fonctions pures et impure. C'est d'ailleurs ce qui en fait sa plus grande différence à première vue avec l'*OCaml*. Cette séparation s'effectue avec l'un des nombreux concepts de la **théorie des catégories** présente dans ce langage. Les **Monades** (On ne citera que le terme de **Monade** dans cette partie, mais les structures **Functor** et **Applicative** visent à la même chose.), est le concept qui permet de confiner toute action impure du reste du code. En effet, lors que par exemple une action *IO (input/output)* doit être faite, elle se trouvera dans la monade `IO`. Si on reprend la fonction `fibo` de la figure 2, et que l'on souhaite afficher le n-ième nombre de Fibonacci avec `n`, un nombre entrée par l'utilisateur on obtient le code de la figure 4.

```haskell
import Text.Read (readMaybe)

main :: IO ()
main = do
    l <- getLine
    let n = readMaybe l :: Maybe Int
    print $ fibo <$> n :: IO()
```

FIGURE 4 – Code *Haskell*, de l'affichage du n-ième nombre de Fibonacci entrée par un utilisateur.

Du code ci-dessus, on peut observer le type `IO`, monade d'entrée et de sortie, mais aussi `Maybe`. Ces deux monades, sont un bon exemple de ce que concept apporte. On pourrait citer la définition donnée par Wikipédia d'une monade figure 5, mais cela ne les ferait comprendre qu'à une mince portion de personne.

'[a Monad is] an endofunctor, together with two natural transformations required to fulfill certain coherence conditions' [7]

FIGURE 5 – Définition d'une Monad selon *Wikipedia*.

On peut voir une monade, comme une structure qui gère les erreurs possibles liées aux **effets de bord**. Ce qui rend ce concept de Monade si important est si on dispose d'une fonction `f :: Int -> Int`, et d'une valeur de type `n :: IO (Int)` l'appelle suivant est une erreur levée à la compilation `f n`. Évidemment, il existe des moyens d'appeler `f`, avec la valeur entière contenu dans `n`. Pour ce faire, on doit avoir recours aux fonctions suivantes :

```
-- Operateur Functor
fmap :: Functor f => (a -> b) -> f a -> f b
(<$>) :: Functor f => (a -> b) -> f a -> f b
-- Operateur Applicative
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
-- Operateur Monad
(>>=) :: Monad m => m a -> (a -> m b) -> m b
(>>) :: Monad m => m a -> m b -> m b
return :: Monad m => a -> m a
```

On remarque alors que toutes ces fonctions font en quelque sorte en conversion vers la monade. Grâce à cette conversion, on peut donc conserver la gestion d'erreur mise en place par ces structures.

Qu'une fine partie des concepts de ce langage n'a été abordée ici. Bien d'autre chose le rende bien différent des langages abordés lors de la licence. Nous aurions pu voir par exemple, ce qu'apporte l'évaluation paresseuse du langage. Où encore, les optimisations agressives fournies par le compilateur *GHC*. De plus, la proximité entre ce langage et la **théorie des catégories**, fait que de nombreux modules introduise des notions pas encore implémenter. On peut notamment citée les *Lens*.

III Resultat

Références

[1] Will Kurt. *Get Programming with Haskell*. Manning Publications, 2018.

[2] Vitaly Bragilevsky. *Haskell in Depth*. Manning Publications, 2021.

[3] Philipp Hagenlocher. *Haskell for Imperative Programmers*. YouTube Playlist. Consulté le 11 juin 2024. 2020. URL : https://www.youtube.com/playlist?list=PLe7Ei6viL6jGp1Rfu0dil1JH1SHk9bgDV.

[4] chshersh. *Haskell Beginners 2022 Course*. YouTube Playlist. Consulté le 11 juin 2024. 2022. URL : https://www.youtube.com/playlist?list=PLOJjn67NeYg9cWA4hyIWcxfaeX64pwo1c.

[5] Brian Lonsdorf. *Professor Frisby's Mostly Adequate Guide to Functional Programming*. Rapp. tech. Consulté le 10 juin 2024. GitBook, 2015. URL : https://mostly-adequate.gitbook.io/mostly-adequate-guide/ch03.

[6] Muthukrishnan. *Fast nth Fibonacci number algorithm*. Rapp. tech. Consulté le 10 juin 2024. 2020. URL : https://muthu.co/fast-nth-fibonacci-number-algorithm/.

[7] Wikipedia. *Monad (functional programming)*. Rapp. tech. Consulté le 11 juin 2024. URL : https://en.wikipedia.org/wiki/Monad_(functional_programming).