Rapport de stage

T.Renaux Verdiere

2023–2024

Résumé

Ce document constitue mon rapport de Stage de huit semaines. Ce stage a été effectué au laboratoire de recherche de l'université ou j'ai effectué cette licence, le GR^2IF (Groupe de Recherche Rouennais en Informatique Fondamentale). Il s'agit d'un laboratoire situé sur le campus du Madrillet de l'Université de Rouen. Fondé en mai 2018, il se concentre sur diverses thématiques telles que les langages formels, la combinatoire algébrique, l'informatique quantique et le génie logiciel.

Table des matières

I Introduction

Ce rapport présente les résultats de mon stage de huit semaines au sein du GR^{2}IF (Groupe de Recherche Rouennais en Informatique Fondamentale), sous la supervision de M. Mignot. Ce stage m'a permis de développer une bibliothèque d'automates en *Haskell*, d'implémenter des tests de propriétés, et de réaliser des benchmarks de performance.

Le rapport commence par une introduction qui présente le contexte du stage. Il se poursuit avec une description des objectifs du stage, notamment le développement de la bibliothèque d'automates et la mise en place d'une interface graphique. La section suivante détaille l'organisation du travail, incluant les réunions hebdomadaires et la planification à l'aide d'un diagramme de Gantt. Les difficultés rencontrées, tant au niveau du stage que des aspects techniques, sont ensuite abordées. Une section est dédiée à la présentation du langage *Haskell*, certaines spécificités de la programmation pure. Les résultats du stage sont ensuite exposés, avec une présentation détaillée de la bibliothèque d'automates, de l'application graphique, ainsi que des phases de tests par propriétés et benchmarks. Les possibilités d'amélioration du projet sont ensuite discutées, suivies par un bilan des compétences acquises et de l'utilisation des connaissances de la licence. Le rapport se termine par l'affinement professionnel que ce stage m'a permis de mettre en place.

II 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 améliorer 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 analyseur (aussi appelé 'parser' en anglais) 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 été 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 trouvées.

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. Cette réunion servait à évoquer les avancées et difficultés rencontrées lors de la semaine. De manière générale, de telles réunions avait un temps variable pouvant aller de 20 minute à pour les plus longues plus d'une heure.

J'ai pour ma part après cette première réunion qui nous a donné 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 correspond 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'homogénéité ou encore savoir s'il est standard (se référer au document technique).
- Jaune : Création d'une conversion d'un automate en *Dot*, qui permet une représentation graphique.
- 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.

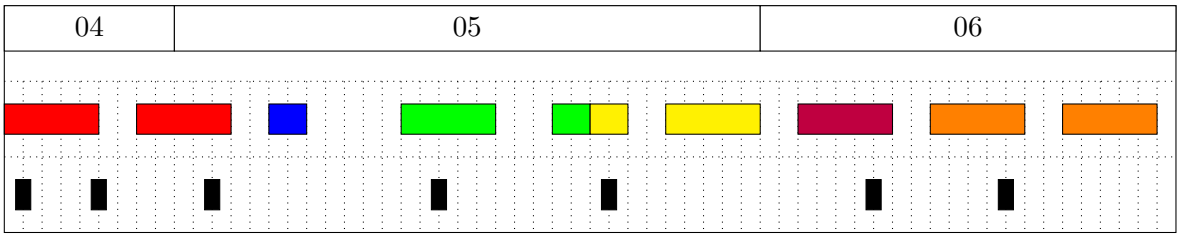

FIGURE 1 – Diagramme de *Gantt* mise en place pour le stage. Les rectangles noirs correspondent au jour de réunion.

2 Difficultés rencontrées

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

2.1 Problèmes liés au stage

Ce stage constitue ma première expérience proche d'un travail en entreprise. De ce fait, j'ai pu rencontrer les problèmes communs. La première semaine, nous étions en télétravail. De nombreux problèmes surviennent par ce mode de travail. Tout d'abord la façon de gérer ces horaires. M. Mignot nous a 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 certaines journées 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 est survenu, celui-ci concernait le matériel. En effet, mon ordinateur portable n'avait selon mon système d'exploitation plus de mémoire vive. J'ai alors du réinstaller un nouvel *OS*. Cependant, je n'avais pas prévue qu'il existait encore des systèmes qui ne se lançait pas sur *systemd*. Sans rentrer dans les détails, ce problème ma pris énormément de temps à régler. C'est un problème, car le gestionnaire de paquets (si on peut le nommer de la sorte) *Nix* (qui est l'outil le plus utilisé pour la création de site en *Haskell* sous le paradigme de la *Programmation fonctionnelle réactive*) doit ce lancer à l'aide de *systemd*. C'est d'ailleurs une des raisons ayant 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 certaines courtes vidéo disponibles sur *Youtube* [3, 4]. De ce fait, malgré avoir commencé assez tôt l'apprentissage de ce langage, comme évoqué dans de nombreuses ressources 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 en place du convertisseur de chaine de caractères vers expression. Les modules utilisés ont été comme conseillé par M. 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 graphe. La documentation est totalement horrible. De plus, le module étant très peu utilisé, il n'existe pratiquement aucun exemple sur internet. J'ai donc dû 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 du site web, il avait été convenu lors d'une réunion, d'utiliser le paradigme de la *Programmation fonctionnelle réactive*. Ce paradigme, se base sur l'idée d'un tableur. Lorsque l'on modifie le contenu d'une case, le reste du tableau peut alors évoluer. C'est sur ce principe que repose la *programmation réactive*. En *Haskell*, il existe de nombreux modules qui permettent la mise en place de cette méthode de programmation. Cependant, celui qui est utilisé dans les interfaces Web, *Reflex*, nécessite l'outil *Obelisk*. Cet outil ma tout d'abord causé des problèmes quant à son installation, puis son uti-

lisation. L'utilisation de *Nix* par celui-ci, oblige à chaque installation, de compiler au moins une fois ce qui a été installé. De ce fait, son utilisation était très longue à mettre en œuvre. Après cela, il y a aussi l'utilisation du module *Reflex*. Bien que ce soit un module *Haskell*, son utilisation requière un apprentissage qui se rapproche de celui d'un langage. Notamment, car nous n'avions jamais vu ce paradigme et donc ces mécanismes m'étaient inconnus. De ce fait, après avoir passé une semaine à essayer d'apprendre le *frp*, je me suis redirigé vers la mise en place d'une application *Gtk* (implémentation bien plus commune, qui permet donc un développement plus rapide).

Enfin la dernière difficulté rencontrée à été la rédaction du rapport formel dans un premier temps. Bien que j'ai 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 à cet exercice lors de la licence, il a été dur de s'y adapter. Je me suis d'ailleurs rendu compte du nombre de relectures nécessaires pour un tel document. Je ne pensais pas qu'il fallait en faire autant. 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. Ayant maintenant l'habitude des rapports de projet avec tous ceux rédigés lors de la licence, je ne pensais pas que celui-ci allait être si dure. La différence entre ce rapport et ceux précédents est surtout la différence de contenu. L'objectif de celui-ci ne vise pas à montrer toutes les facettes du projet sur lequel j'ai travaillé, mais plutôt la manière dont j'ai travaillé dessus et ce que j'ai appris.

III *Haskell*

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 vu 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'est 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

```haskell
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 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 évoqué plus tôt, *Haskell* met en place une séparation hermétique entre les fonctions pures et impures. C'est d'ailleurs ce qui en fait sa plus grande différence à première vue avec *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. 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 utiliser les monades, comme un outil gerant 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 contenue 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 une 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'autres choses le rende bien différent des langages abordés lors de la licence. Nous aurions pû 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 introduisent des notions pas encore implémentées. On peut notamment citée les *Lens*.

IV Résultat

1 Bibliothèque d'automate et application graphique

Dans cette section, je présenterai les résultats finaux de mon travail, en détaillant les étapes de vérification et de tests de performances. Nous examinerons les différentes parties du projet, y compris l'implémentation de la bibliothèque d'automate et l'application graphique qui a été développée.

1.1 Bibliothèque d'automate

L'objectif de ce stage était la réalisation d'une bibliothèque complète d'automate. J'ai donc mis en place, la conversion d'une expression vers un automate de *Glushkov*. La conversion d'un fichier *JSON* vers un automate. La représentation sous forme de graphique utilisant le langage *Dot*. Ou encore des tests et conversions de diverses propriétés d'automates, comme la standardisation ou encore les orbites maximales et leurs propriétés.

J'ai mis en place une première version de cette bibliothèque qui utilisait la structure de la figure 6. Le point crucial de cette implémentation est la fonction `delta`. Celle-ci rend la structure identique à la spécification mathématique d'un automate. Les opérations d'ajout et de suppression sont implémentés par l'agrandissement de la fonction.

Le plus grand problème de cette implémentation réside dans le calcule des orbites. N'aillant de structure à parcourir, il a donc fallu convertir cet automate vers un graphe puis calculé ces orbites. C'est après avoir implémenté cette fonction que j'ai donc pris la décision de mettre en place une autre implémentation. Une comparaison de performances des deux implémentations et de la validité du développement d'une autre implémentation se trouve dans la section 2.2.

```
data NFAF state transition = NFAF
  { sigma   :: Set.Set transition
  , etats   :: Set.Set state
  , premier :: Set.Set state
  , final   :: Set.Set state
  , delta   :: state -> transition -> Set.Set state
  }
```

FIGURE 6 – Code du type `NFAF`.

L'implémentation que j'ai mise en place après celle-ci est celle de la figure 7. La principale contrainte du type `NFAF` était l'impossibilité de parcourir l'automate. J'ai

donc eu l'idée d'utiliser un graphe du type *fgl* pour ce faire. Ce module fournissant une implémentation efficace à l'aide de *Patricia Tree* appellé aussi *Arbre radix*. Les éléments de types `Int` permettent de faire la correspondance entre le nœud du graphe et l'état. En effet, dans un graphe du module *fgl*, un nœud est composé d'un 'tuple' entier label. Pour améliorer les performances du type, des entiers sont donc utilisés pour les ensembles d'états premier et final par exemple.

Le plus grand défaut de cette implémentation réside dans l'utilisation mémoire plus grande pour ce type. La mise en place d'un graphe où chaque transition est présente en mémoire. Alors que pour l'implémentation utilisant une fonction, seul les cas de tests de la fonction grandirons ce qui aura un impact bien moindre sur la mémoire.

```
data NFAG state transition = NFAG
  { sigma   :: Set.Set transition
  , etats   :: Map.Map state Int
  , premier :: Set.Set Int
  , final   :: Set.Set Int
  , graph   :: Gr.Gr state transition
  , lastN   :: Int
  }
```

FIGURE 7 – Code du type `NFAG`.

1.2 Application graphique

L'application permet d'obtenir un automate à partir d'une expression entrée par l'utilisateur. On peut importer un automate sous le format *JSON*. Le format que doit vérifier le fichier est celui présenté sur la figure 8. On peut aussi choisir d'afficher l'automate avec les orbites affichées, et ce, à l'aide du menu déroulant que l'on obtient en appuyant sur le bouton options que l'on peut voir sur la figure 9.

```
{
  "nodes": [0, 1 ..],
  "first": [0, ..],
  "final": [1, ..],
  "transitions": [[0, 1, "a"], ..]
}
```

FIGURE 8 – Figure montrant le format d'un automate sous forme *JSON*.

Une fois l'automate obtenu par la transformation d'expression en automate ou bien par importation d'un fichier *JSON*, on peut obtenir des informations sur les ensembles d'états de l'automate. Ces informations sont les suivantes :

- Est-ce que cette ensemble est une orbite.
- Est-ce que l'orbite est stable, transverse, fortement stable ou bien encore hautement transverse.
- Les portes d'entrée ou de sortie de ce groupe d'états.

Pour choisir ce groupe d'états, deux choix s'offrent à l'utilisateur. Le premier est l'input manuel et le deuxième, le menu déroulant qui répertorie toutes les orbites maximales de l'automate actuellement choisi. De plus, une fois le groupe d'états choisis, on peut l'observer dans l'image à gauche des propriétés testées. Voir la deuxième capture de la figure 9 pour une démonstration de ces fonctionnalités.

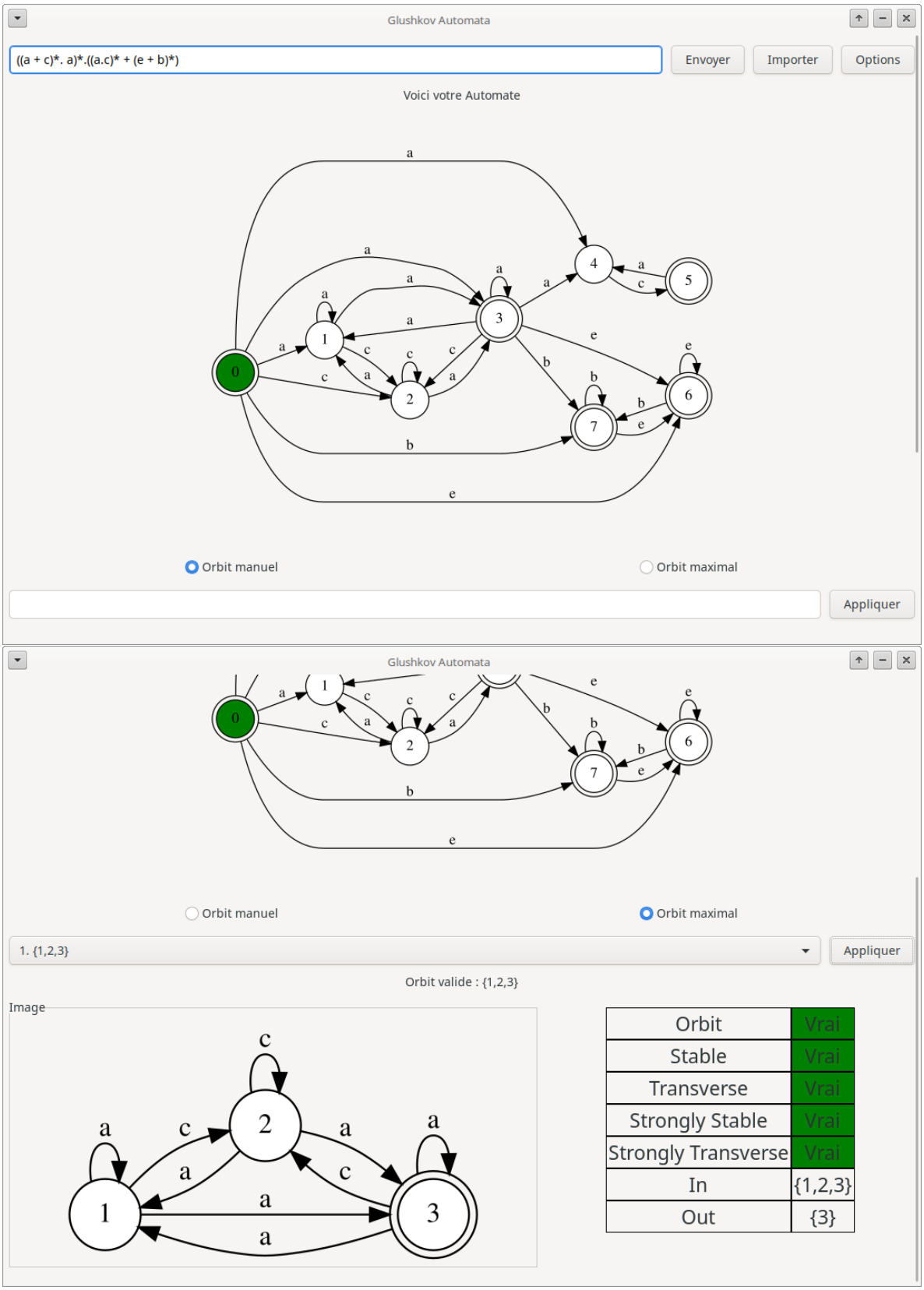

FIGURE 9 – Capture de l'application graphique.

2 Phases de tests

Pour ce projet, la phase de test, je l'ai mis en place à la fin de ce stage. En effet, cette partie n'avait été prévue d'être réalisée que s'il restait assez de temps pour ce faire. Les réunions hebdomadaires servaient entre autre à jouer en partie ce rôle. De plus à chaque ajout de fonctionnalités, j'essayai d'effectuer certains teste. Malgré cela, j'ai pû trouver le temps de mettre en place deux types de tests. Nous commencerons par voir la façon dont j'ai mis en place des tests par propriétés. Puis nous verrons les testes de performances mises en place entre les deux implémentations d'automate.

2.1 Test par propriétés

Les tests par propriétés, *'property testing'* en anglais. Est une méthode de tests logicielle qui visent à tester les propriétés d'un résultat et non pas à comparer le résultat obtenu avec celui espéré [8]. De plus, cette méthode ne nécessite pas d'avoir une autre implémentation pour comparer les résultats ou même de calculer à la main les résultats des opérations que l'on veut tester (Cette méthode est celle communément utilisé, des *tests unitaires*).

Outre cela, cette méthode de test convient extrêmement bien à la production d'objets mathématiques. Notamment en ce qui concerne les automates, ils existent de nombreuses propriétés connues qui permettent de faciliter ces tests. Plus précisément, on peut par exemple pour les *automates de Glushkov* se référer à l'article *Characterization of Glushkov Automata* de Pascal Caron et Djelloul Ziadi [9]. Bien évidemment, cette méthode ne montre pas que le programme fait exactement ce qu'il doit faire. Il faudrait pour cela effectuer des preuves formelles mathématiques, telles que celle vue en cours d'algorithmique lors de cette licence (par le biais de la logique de *Hoare* par exemple). Seulement, ces preuves se révélant très longues à écrire, les tests par propriétés suffisent.

En *Haskell*, c'est la bibliothèque *QuickCheck* qui permet de mettre en place ce type de test. Une fois des propriétés définies, une génération aléatoire de cas de test sera effectuée, et ça sera sur ces cas que les propriétés se verrons testées. Pour générer ces cas de test aléatoire, il faut que les objets en question dérive la classe `Arbitrary` du module. Le terme 'dérivé' signifie ici que notre classe doit implémenter la méthode `arbitrary :: Gen a` avec `a`, le type que l'on veut tester. C'est grâce à cette fonction `arbitrary` que seront généré de façon aléatoire les objets sur lesquels seront testé les propriétés. Enfin, pour tester ces propriétés, il faut appeler la fonction `quickCheck` qui prendra en paramètre une fonction qui doit avoir ce type `prop :: Arbitrary a => a -> Bool`. Cette fonction effectuera une génération de 100 objets et testera la fonction `prop` sur ces objets.

En ce qui concerne la mise en place des tests des deux implémentations de la bibliothèque d'automates, la première étape a été l'implémentation de la méthode `arbitrary` pour les types d'automates de la bibliothèque. Pour ce faire, la création d'un automate

se fait par la génération d'une liste d'état et d'une liste de transition par le biais du module *QuickCheck*. Ce ne sont pas les types `NFAG` et `NFAF`, qui implémente cette méthode, mais des types enveloppes. Cela est dû à la seule utilité de ce type pour ces tests. Une convention du langage n'autorise pas l'implémentation d'une classe par un type hors de son fichier d'implémentation. Et il n'y a aucun intérêt à ce que ces deux types implémentent `arbitrary` or des tests. Finalement les propriétés testées sont celle de la table 1.

Propriété	**Description**
`propAddState`	Vérifie que l'ajout d'un état.
`propAddTransition`	Vérifie l'ajout d'une transition.
`propRemoveState`	Vérifie la suppression d'un état.
`propRemoveTransition`	Vérifie la suppression d'une transition.
`propDirectSucc`	Vérifie les successeurs directs d'un état.
`propDirectPred`	Vérifie les prédécesseurs directs d'un état.
`propStandard`	Vérifie les propriétés d'un automate standard.

TABLE 1 – Résumé des propriétés testées sur les automates.

Enfin, les testes sur la partie de transformation d'expression régulière en automate de *Glushkov* se fait sur l'implémentation `NFAG` (se référer à la section suivante pour les raisons de ce choix.). Comme pour la partie précédente, il faillait tout d'abord générer des expressions de manière aléatoire. Pour ce faire, j'ai utilisé le 'parser' développé, permettant à partir d'une chaine de caractères d'obtenir l'expression correspondante. Il m'a donc fallu créer une chaîne représentant une expression, et cela, de manière aléatoire. J'ai alors utilisé les fonctionnalités du module *QuickCheck* pour cela. Après la génération de ces expressions régulière, une conversion vers un automate s'effectuer à l'aide de l'algorithmique de *Glushkov*. C'est cette convention qui se voit être testée par les propriétés définies dans l'article [9]. Avec notamment la définition d'orbite et d'orbite maximale et des propriétés qu'ils doivent respecter dans le cas d'un automate de *Glushkov* (ces propriétés sont les suivants, un automate de *Glushkov* doit être **standard**, toutes les orbites maximales de cette automate doivent être **fortement stable** et **hautement transverse**).

On obtient alors les résultats de la figure 10 lors d'un appelle à ces tests. Ce qui certifie la validité des implémentations faites.

```
$ stack test
Testing NG.NFAG implementation...
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
Testing NF.NFAF implementation...
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.

Testing Gluskov properties...
+++ OK, passed 100 tests.
```

FIGURE 10 – Résultat des tests de propriétés des implémentations d'automates.

2.2 Benchmark des implémentations d'automates

Pour évaluer les performances des deux implémentations d'automates **NFAG** et **NFAF**, l'utilisation du module *Criterion* a été faite. Pour rappel, la principale différence entre ces deux implémentations est que **NFAG**, utilise un graphe du module *fgl* tandis que **NFAF** utilise une fonction. Nous commencerons par voir comment ce module de Benchmark fonctionne, puis nous comparerons les deux implémentations.

Le module *Criterion* permet de mesurer précisément le temps que prennent différentes parties du code à s'exécuter. Il effectue de nombreuses répétitions des tests pour obtenir des mesures fiables et fournit des analyses statistiques sur les résultats, comme les moyennes et les écarts-types. De plus, ce module permet d'exporter les résultats des tests sous formes d'une page *HTML*, avec notamment des graphiques. Le seul point négatif de ce module est que seul les temps d'exécutions sont mesurés, la mémoire utilisée ne l'était pas.

Pour tester les performances des deux implémentations les tests suivants ont été mis en place :

- **Ajout d'états** : Nous avons mesuré le temps nécessaire pour créer un automate avec un nombre fixe d'états (1000 états).

- **Ajout de transitions** : Nous avons évalué les performances de l'ajout de transitions à un automate. Cette opération vise à ajouter une transition entre chaque nœud de l'automate (a titre informatif l'appelle de cette fonction sur un automate à 2000 états ou plus utilise tout met huit giga de RAM. Ce qui éteint mon ordinateur).
- **Suppression d'état et de transitions** : Il existe la réciproque des deux testes précédents, mais dans la suppression.
- **Teste d'orbites maximales** : Après la création d'un automate de taille prédéfinie (dans les résultats présentés plus tard, c'est un automate de 2000 états qui a été conçu) de façon aléatoire, un test visant à savoir si chaque orbite maximale et fortement stable et hautement transverse sera effectué.

Les résultats de ce test de performances sont en autre ceux de la figure 11, un rapport bien plus détaillé (celui produit par le module *Criterion*) peut être trouvé sur ce lien Rapport du Benchmark. Sur cette capture, on voit deux couleurs, celle orange correspondant au type `NFAF` et le bleu au type `NFAG`. On peut alors observer que la deuxième implémentation est bien plus performante que la deuxième. Pouvant même aller jusqu'à deux fois plus rapide selon les tests.

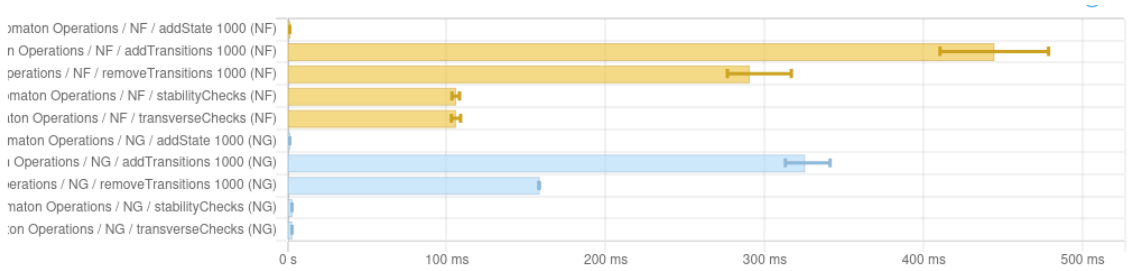

FIGURE 11 – Capture d'une partie du rapport de Benchmark des implémentations.

3 Les possibilités d'amélioration

Dans cette section, nous discuterons des différentes possibilités d'amélioration pour ce projet, visant à augmenter ces fonctionnalités, sa facilité d'utilisation et sa documentation.

Une possibilité d'amélioration consiste à développer un site web, particulièrement en utilisant la *Programmation Réactive Fonctionnelle (FRP)*, comme mentionné dans les difficultés rencontrées. Avec plus de temps, il aurait pû être possible de mettre en place cela et donc de répondre au problème rencontré.

Une autre amélioration serait de mettre en place une interface graphique plus avancée, permettant par exemple la création d'automates en temps réel. Que ce soit par l'ajout d'une autre page, avec un système qui pourrait être proche de celui de *GeoGebra* par exemple. Cela permettrai d'utiliser des automates autrement que par la

rédaction d'un fichier *JSON* qui peut se révéler très long en fonction de la structure de celui-ci.

Il serait également bénéfique d'ajouter davantage d'opérations sur les automates. L'ajout d'algorithmes d'homogénéisation, de détermination du langage reconnu par l'automate ou encore des opérations de réduction d'état. Avec cet ajout de fonctionnalités, on obtiendra alors une bibliothèque hautement réutilisable. À voir encore s'il n'existerait pas de structure plus performante que celles présentées ici.

Une spécification plus détaillée pourrait être fournie, en listant tous les algorithmes utilisés et en introduisant des exemples pour faciliter la compréhension du rapport. De plus, ce rapport technique étant l'un des premiers que je rédige, la rédaction d'autre permettrait l'amélioration de celui-ci par le biais de l'expérience acquise.

Il serait également utile de fournir une documentation complète sur le code des deux implémentations. Bien que la documentation en *Haskell* diffère des autres langages, le module *Haddock* pourrait être utilisé pour générer cette documentation. Il génère un ensemble de page *HTML* proche de ce que l'on peut retrouver le site référence *Hackage*.

Enfin, l'ajout d'un benchmark de mémoire utilisé serait une amélioration significative, bien que cela soit plus complexe que les benchmarks temporels et ne soit pas pris en charge par le module *Criterion*. Il existe cependant des alternatives comme le module *weigh* mais qui ne produisent pas de rapports tels que ceux produits par le module *Criterion*.

V Bilan

1 Ce que j'ai acquis pendant le stage

Durant ce stage de huit semaines en laboratoire, j'ai acquis de nombreuses compétences techniques. J'ai approfondi mes connaissances en *programmation fonctionnelle* et notamment en *Haskell*, notamment sur la programmation pure et l'utilisation des monades. J'ai développé une bibliothèque d'automates, mis en place des tests par propriétés et réalisé des benchmarks de performance. Cette expérience m'a également permis de me familiariser avec des outils et des modules spécifiques au *Haskell*, malgré les défis posés par une documentation qui sort des normes apprises lors de la licence.

En termes de bénéfices, j'ai amélioré ma gestion du temps et des priorités grâce à l'utilisation de technique de priorisation tel que l'utilisation d'un diagramme de *Gantt*. J'ai aussi appris à surmonter les obstacles du télétravail, notamment en gérant mes horaires et en résolvant des problèmes matériels. En somme, cette expérience a été extrêmement enrichissante, confirmant mon intérêt pour l'aspect théorique de l'informatique et m'apportant une vision claire des attentes et exigences du secteur, par le biais de la rédaction d'une spécification formelle.

2 Les connaissances de la licence qui m'ont été utile

Les savoirs acquis durant ma licence en informatique m'ont été très utiles tout au long de ce stage. Mes connaissances en *programmation fonctionnelle*, acquises lors des cours sur *OCaml*, m'ont fourni une base solide pour aborder ce nouveau langage. De plus, les compétences en algorithmique et en structures de données m'ont aidé à développer et optimiser les algorithmes nécessaires pour la bibliothèque d'automates. Bien évidemment, la matière de théorie des langages ma aidé quant à la compréhension de notions déjà vue auparavant.

Les projets pratiques réalisés durant ma licence m'ont préparé à aborder des possibles problèmes techniques ainsi que la gestion du temps. La rigueur avec laquelle on nous a enseigné les langages *C* et *Java*, ma poussé à toujours aller plus loin dans l'apprentissage du *Haskell*. Avec lecture systématique des documentations, mais aussi d'essayer de comprendre au maximum ce qu'il se passait. De plus, il se trouve que certains projets que nous avons dû produire pour la licence se trouve très près de certaines parties de celui-ci. Le parser d'expression régulière utilise les modules *Happy* et *Alex*, ces deux modules sont basés sur *Bison* et *Lex* ce qui a rendu leur utilisation bien plus simple. En effet, lors du projet de compilation, nous avions du utiliser *Bison* et *Lex* afin de mettre en place un compilateur assez simple.

3 Affinement du projet professionnel

Cette expérience m'a définitivement aidé à affiner mon projet professionnel et mon projet d'étude. Travailler sur un projet réel en *Haskell* et contribuer à une bibliothèque d'automates m'a confirmé mon intérêt pour l'aspect théorique de l'informatique. Le langage *Haskell* en est un parfait exemple, avec ces bases dans la théorie des catégories. Je prévois d'ailleurs de commencer mon apprentissage de ce sujet durant les vacances qui vont suivre ce stage.

Elle m'a également permis de mieux définir mes objectifs professionnels. Ce stage m'a conforté dans l'idée de devenir chercheur. Cette expérience m'a conforté dans mon choix de master, celui *ITA*. Travailler sur des projets de recherche concrets, analyser les résultats et proposer des améliorations m'ont donné un aperçu de la rigueur et de la créativité nécessaires dans le domaine de la recherche. Le stage m'a aussi permis d'échanger avec un étudiant en 2e année de ce même master, ce qui a enrichi ma vision de celui-ci.

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).

[8] Wikipedia. *Property testing*. Rapp. tech. Consulté le 16 juin 2024. 2024. url : https://en.wikipedia.org/wiki/Software_testing#Property_testing.

[9] Pascal Caron et Djelloul Ziadi. "Characterization of Glushkov Automata". In : *Theoretical Computer Science* 233.1-2 (2001), p. 75-90.