Rapport de Stage de L3 Informatique
ISTIC - Université de Rennes
Année 2023 - 2024

Détection de données incomplètes dans une base de données botanique

Charlotte Thomas
Stage effectué à Inria Rennes, dans l'équipe EPICURE
Supervisé par M. Simon Castellan
Du 13/05/2024 au 12/07/2024

Table des matières

I Présentation du stage

Mon stage de fin d'année - dans le but de l'obtention de ma Licence d'Informatique - a été effectué en programmation dans un laboratoire de recherche public, ce choix a été fait en raison d'une appétence pour le secteur public ainsi que par l'admiration pour la recherche.

I.1 Projet

Le projet sur lequel ce stage fait partie s'appelle *Back To The Trees* dont l'objectif est la génération de *clef* de détermination botanique. Une *clef* de détermination est un arbre de décision permettant de retrouver l'espèce à laquelle appartient une plante [1]. Les noeuds de cet arbre correspondent à des critères morphologiques tels que la forme des feuilles ou la couleur des pétales sur les plantes à fleurs. Fig. 1 présente un exemple de clef

Le projet s'inscrit dans une envie de démocratiser les connaissances botaniques par le travail conjoint avec des associations qui font des balades botaniques, on a ici de faibles *collections* d'environ 200 espèces contre une flore nationale qui en contiendrait 5000, d'où le besoin d'un générateur pour avoir des clefs pour chaque collection. Les questions posées utilisent un vocabulaire compréhensible par le grand public tout en assurant la validité des connaissances transmises. Il s'inscrit dans un domaine plus large de reconnaissance morphologique, c'est à dire des caractéristiques physique des plantes pour reconnaître l'espèce de plante que l'on vise.

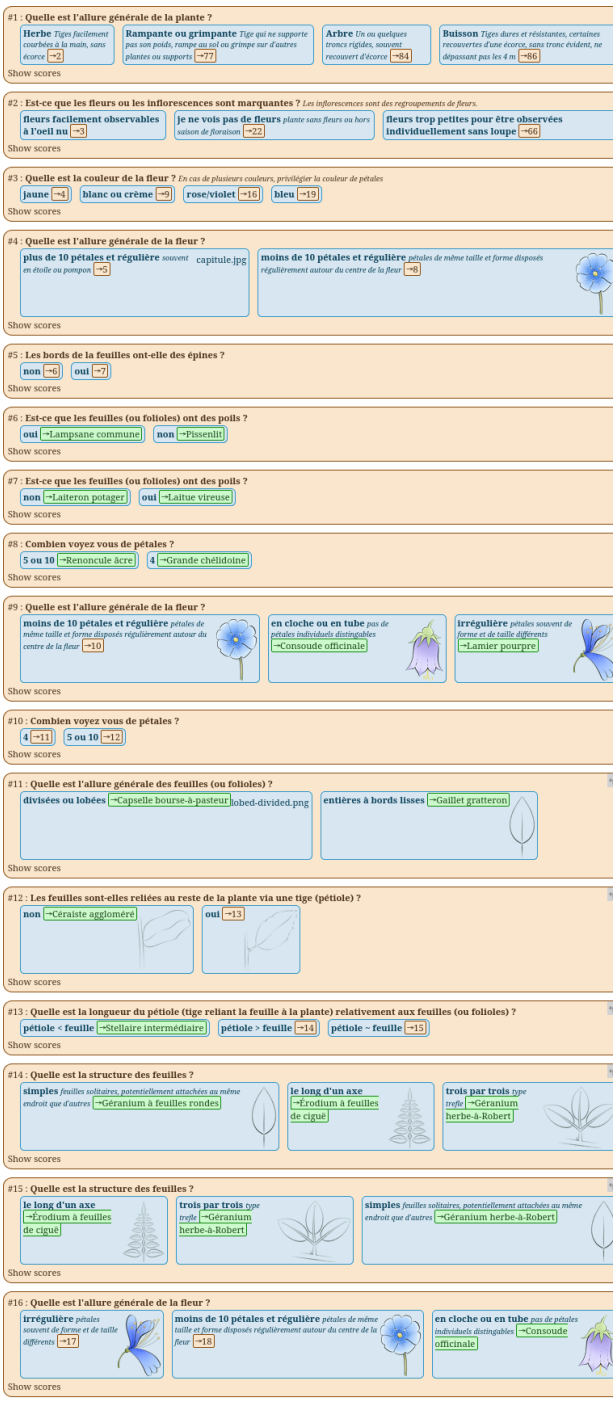

Fig. 1. – Un exemple de clef

Le code [2] (en OCaml) de cet outil est entièrement ouvert sous licence libre (plus exactement la GNU General Public Licence 3-or-later). Le projet est financé par une action exploratoire *back to the trees* Inria et une ANR[1] **Flores**.

[1] Agence Nationale de Recherche

M. Simon Castellan, mon encadrant de stage, est porteur du projet et c'est dans ce projet que *Mme Aurore Alcolei* qui m'a aussi aidée et encadrée, a été recrutée en tant qu'ingénieure de recherche. Des botanistes travaille également sur ce projet pour s'assurer de la validité du savoir botanique transmis.

I.2 Structure d'accueil

Ce stage a eu lieu au *Centre Inria de l'Université de Rennes*, un des centres de Inria, un institut national réparti sur tout le territoire qui se charge de la recherche en informatique au sens large. Les domaines étudiés vont de l'informatique formelle théorique aux réseaux de neurones en passant par la science de l'ingénieur avec l'impression 3D.

Le centre est aussi divers au niveau des instituts et centres de recherche le centre en fait parti, il y a l'IRISA,[2] qui est une UMR[3] dont fait partie Inria. et le CNRS[4] sans compter les partenariats entre UMR et entre chercheurs qui permettent par exemple dans le cadre du projet de ce stage la collaboration entre des chercheurs en botanie et en informatique.

Le centre est découpé en plusieurs services ainsi que des « *équipes-projets* », certaines sont communes à Inria et à l'IRISA comme l'équipe où ce stage a eu lieu, *EPICURE*.

Emprunté directement depuis le site présentant l'équipe [3].

« *The goal of the EPICURE project is to contribute with semantics-based methods for producing safe and secure software* ».

EPICURE se présente donc comme une équipe de recherche sur la sécurité, la sémantique et la fiabilité des système en s'appuyant sur la sémantique. On y trouve beaucoup de recherche en langages formels et en sémantique des langages dont notre projet est un exemple de théorie des langages de programmation.

Il y a beaucoup de personnels travaillant dans cette équipe, que ce soit nos professeurs de notre classe ou du personnel non enseignant (uniquement chercheur), comme le responsable de ce stage.

II But du stage

Ce stage a un double but, l'un de générer les tests de regression (le framework, pas les tests) et l'autre beaucoup plus important est la détection des données manquantes dans la base de données morphologiques sur laquelle se repose le projet.

II.1 Structure du projet

Comme mentionné en introduction, le logiciel plantinator permet la génération automatique de clef de détermination botanique.

Pour cela on utilise 3 modules :

- La base de donnée morphologique remplie par les botanistes avec les caractéristiques des plantes.
- L'algorithme de génération de clef automatique.
- Le traducteur

La base de donnée contient des collections qui contiennent les fiches espèces ayant chacune le même schéma de descriptions de trait morphologique. Une clef de détermination est ainsi générée pour chaque collection qui correspondent, comme on l'a dit dans l'introduction, aux differents lieux pour lesquels on veut éditer des clefs.

[2] *Institut de Recherche en Informatique et en Système Aléatoire*

[3] *Unité Mixte de Recherche*

[4] *Centre National de la Recherche Scientifique*

Le code de *plantinator* est découpé en plusieurs partie distinctes. La partie `core` s'occupe de tout ce qui est le coeur du programme et commun à tout le reste. Ensuite on a le `cli` qui héberge le code de l'éxecutable `plantinator`. La partie `web` est le site client (OCaml compile vers du JavaScript qui est utilisé pour le client) et enfin le `server` contient le code du serveur web.

Les botanistes remplissent des données morphologiques hors de portée du grand public, et on a besoin que n'importe qui puisse comprendre les clefs de détermination, on a donc besoin d'un traducteur.

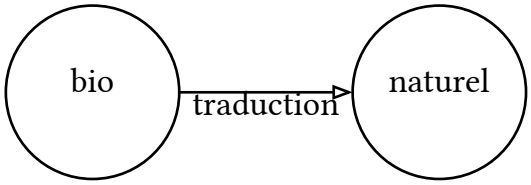

Fig. 2. – Rôle de la traduction

Par exemple sur la forme de la corolle la traduction va de cette liste

```
Forme de la corolle:
plane
rotacée
hypocratériforme
unguiculate
tubulée
bol
cloche
infundibuliforme
urcéolée
ligulée
papilionacée
labiée
```

à cette question (avec les valeurs)

```
Quelle est l'allure générale de la
fleur ?
- capitules en étoile ou pompon,
souvent avec une multitude de pétales
apparents (> 10)
- moins de 10 pétales et régulière
pétales de même taille et forme
disposés régulièrement autour du
centre de la fleur
- en cloche ou en tube pas de pétales
individuels distingables
- irrégulière pétales souvent de forme
et de taille différents
```

Le domaine dans lequel ce stage se place est dans la programmation / théorie des langages, il s'agit de développer des fonctionnalités supplémentaires au logiciel *plantinator* en *OCaml* appartenant aux langages dit *fonctionnels*.

II.2 Tâches demandées

II.2.1 Première tâche : le framework de tests

Pour mieux appréhender le code du projet et le comprendre, la première tâche de ce stage a été de créer un *framework* de test de régression pour notre traducteur, pour que nous puissions ensuite faire des modifications à ce dernier et s'assurer que les modifications n'aient pas fait régresser le traducteur. Ce qui se traduit par l'ajout d'une commande à plantinator, `test --regression-dir=directory`. Cette commande doit tester tous les fichiers qui sont dans le dossier `directory` qui finissent par `.ml` (qui sont des programmes de traductions), si le fichier `.result` (qui contient la description de l'observation correspondant aux programmes de traductions) du même nom existe, la commande compare et dit si le test est passé ou non, si non elle créé un fichier finissant en `.result` contenant la description de l'observation correspondant au programme de traduction du même nom.

Si le fichier `.ml` a une erreur de syntaxe alors le fichier n'est pas lisible par le programme et le test est alors marqué comme invalide.

Il est important de noter qu'ici la tâche n'était pas *d'écrire* des tests mais bien d'é-

crire *le framework* de tests qui va permettre à ces derniers d'être éxécutés.

II.2.2 Deuxième tâche : La détection de données manquantes

Le but est de trouver les données qui sont manquantes dans la base de données morphologiques pour générer des questions dans la clef.

En effet la base de donnée peut être incomplète (les fiches espèces ne sont pas forcément remplies à 100%), cela n'empêche pas la génération de la clef car une valeur par défaut est utilisée, mais cela affaiblie la qualité de la clef (puisque cette valeur est arbitraire et non issue de la réalité botanique). On veut donc détecter les données manquantes.

Par ailleurs, toutes les données morphologiques ne sont pas forcément utilisées dans la traduction. Un exemple que l'on peut donner est le nombre d'étamines, qui est impossible à voir à l'oeil nu, donc il n'est pas pertinent pour le grand public.

La question logique qu'on en tire est donc la suivante: *Comment s'assurer de remplir les traits qui correspondent à une question en premier*. Remplir les données morphologiques est un travail long et fastidieux et l'on aimerait prioriser les traits utilisées par le traducteur dans le remplissage de la base de donnée morphologique.

Nous devons instrumenter le traducteur pour générer un fichier CSV qui contient chaque espèce d'une *collection* donnée. On calcule ainsi la liste des champs vides correspondant à la question afin d'être remplie par les botanistes. Ce fichier CSV doit avoir la trace suivante.

```
nom;tpath;question;valeurs;valeur
```

Liste 1. – Trace du fichier CSV demandé

La première colonne est simplement le nom en latin de l'espèce, la seconde est le *chemin* (un TPath) dans le type de la question, la suivante est la question, la quatrième est l'ensemble des valeurs possibles existant pour être remplis et la dernière est la valeur à remplir par le botaniste.

Un TPath est juste un *chemin* dans le type. Par exemple si un type est le suivant

```
Record (Flower)
  [Sum (Shape
    (Sum
      (Bent (Record [yes, no]),
      Straight (Record [yes,no])
    )),
    Color (Record [blue,green,red]))
  ]
```

Liste 2. – Exemple de type

Un TPath possible est le suivant

```
flower.shape.bent.yes
```

Liste 3. – Exemple de TPath

Pour pouvoir l'importer dans notre projet, en effet nous voulons aussi créer une fonction inverse qui importe les données remplies par les botanistes et les appliquent à la base de donnée correspondante.

III Ce qui a été réalisé

Le stage a principalement porté sur le code de `plantinator` écrit en OCaml. Le travail a été très incrémental nous allons voir les différentes étapes passées.

III.1 Les tests

La partie sur les tests a été réalisée en un peu plus d'une semaine avec au total une quinzaine d'heure de code « pure (auxquelles il faut ajouter le temps de réflexion ainsi que de nombreux tests).

III.1.1 La première approche : comparaison de chaînes de caractères.

Une première approche était de calculer avec une fonction `assert_equals` (de type `Observation.t -> Typ.t -> string -> unit` qui prend donc comme paramètre une observation, un type et un nom de fichier et ne renvoie rien) une *observation*, celle du fichier test `test.ml` (le fichier de traduction comme indiqué précédemment) - une observation est une structure de donnée qui correspond aux descriptions morphologique des plantes. Elles suivent un type commun - et ensuite je comparais ligne à ligne la sortie en chaîne de caractères en utilisant la fonction `to_string` du module *Observation* avec le fichier `test.result` qui est généré la première fois qu'on appelle un test donné, c'est une comparaison ligne à ligne de chaînes de caractères qui demandait beaucoup de filtres et d'appels à la fonction `replace` du module *String*.

La fonction fonctionnait et j'avais ajouté un peu de couleur pour rendre le tout plus joli.

Le problème principal avec ce premier jet était qu'il comparait des chaînes de caractères, la comparaison pourrait changer si on change la méthode `to_string` ce qui demanderais de refaire les fichiers `.result` des tests. Ce n'était pas robuste.

Ensuite la fonction demandait beaucoup de ressource et le style de code n'étaient pas très beau on enchaînait les filtres sur les listes.

III.1.2 Le second essai : La comparaison native directe

Heureusement une solution était disponible: utiliser l'opérateur `(=)` natif d'OCaml.

Après concertation avec mes encadrants de stage, il a été décidé de comparer direc- tement les observations, une observation serait lue depuis le fichier test.result et l'autre observation serait générée depuis le fichier `test.ml` ensuite on fait un bête et méchant = et le tour est joué. Cette technique avait le mérite de simpli- fier beaucoup l'approche, puisqu'on n'avait pas à comparer ligne à ligne des chaînes de caractères mais juste une comparaison na- tive d'OCaml.

Sur le papier cette technique était impossible à rater.

Cependant j'avais des soucis : deux observations que je venais de générer (donc je savais qu'elles étaient égales) et où je pou- vais afficher les chaînes de caractère géné- rés par to_string ne passaient pas. Je ne pouvais pas trouver d'où venait le problème.

Une solution venait alors pour aider, les *ppx yojson*.

III.1.3 Le json à la rescousse

Qu'est-ce que sont les *ppx* en OCaml ? Ce sont des *extensions de préprocesseurs* (qui sont appelées *avant* la compilation) et elles permettent beaucoup de choses très intéressantes en OCaml mais celle qui nous intéresse est la ppx de *yojson* qui permet d'utiliser le format JSON (*JavaScript Object Notation*) en OCaml. Et grâce à ces fameuses ppx on peut *dériver* une observation en json c'est à dire afficher une *représentation* en *json* d'une observation

```
"Sum",
   [
     {
       "explanations": [],
       "contributors": []
     },
     [
       {
         "index": 0,
         "weight": 0.5,
         "observation": [
           "Record", []
         ]
       },
     ]
   ]
```

Liste 4. – Extrait de la représentation en JSON d'une observation obtenue depuis le fichier test.result

```
"Sum",
   [
     {
       "explanations": [],
       "contributors": [""]
     },
     [
       {
         "index": 0,
         "weight": 0.5,
         "observation": [
           "Record", []
         ]
       },
     ]
   ]
```

Liste 5. – Extrait de la représentation en JSON d'une observation obtenue depuis le fichier test.ml

Liste 4 nous montre une observation qui a été passée par ce *ppx yojson* obtenue par le fichier result tandis que Liste 5 nous montre une observation qui a été passée par ce *ppx* obtenue par le fichier `.ml`. La différence est minime, c'est juste dans `contributors` l'un a une liste vide et l'autre une liste vide avec une chaîne de caractères vide à l'intérieur. Différence insignifiante les deux sont pourtant équivalent ? Mais c'est assez pour que l'opérateur `(=)` réponde qu'elles sont différentes.

Une fois ceci corrigé par une fonction qui enlève les chaînes vides à l'intérieur des listes l'égalité fonctionne parfaitement.

III.1.4 Conclusion des tests

Le framework de tests n'existait pas avant ce stage et il sera très utile dans le futur du projet pour permettre de faire des tests de regression. Il n'a pas été demandé d'écrire les tests de régression en eux mêmes, ils seront écris ultérieurement par d'autres personnes du projet.

Mes encadrants de stage étaient satisfaits du travail qui a été fait sur le framework de tests, et il va être utilisé sous peu sur *plantinator* pour faire des tests de régression. Dans l'ensemble le cahier des charges qui a été spécifié en (II) a été respecté et fini. Toutes les fonctionnalités sont présentes et fonctionnelles.

Fig. 3 présente un exemple de test qui passe, un test qui est invalide et un test qui rate, qui sont les trois demandes qui ont été émises par les encadrants du stage.

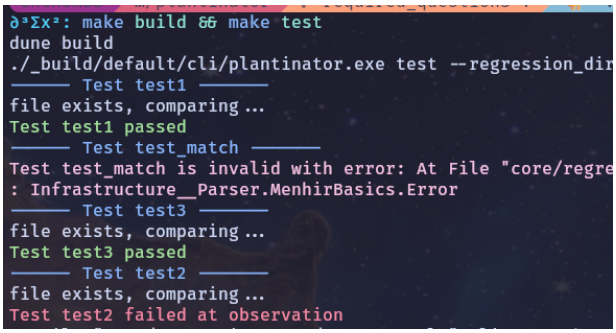

Fig. 3. – Exemple du framework de tests demandé

III.2 Caractéristiques manquantes

Pour aider les botanistes travaillant sur le projet, nous avons eu l'idée de leur faire

remplir des tableurs avec les valeurs manquantes (comme indiqué en II) et d'utiliser ce tableur pour mettre à jour la base de donnée.

On veut donc détecter les données manquantes lors d'une traduction et produire un fichier CSV qui les retranscrit.

III.2.1 La première approche : Propager des TPath

Pour séparer les fonctionnalités, on répartit le travail en deux:

- On cherche à instrumenter l'évaluateur pour qu'il renvoie une liste correspondant aux données incomplètes (i.e. la liste de *TPath*) dans la l'observation évaluée.
- On construit un module tabular qui gère l'encodage en CSV de cette liste

L'évaluateur se distille à ces trois fonctions :

- `Pattern#matchp` dont le type de sortie est `float * Observation.t Env.t`
- `Eval#expression` avec comme type de sortie `Observation.t maybe`
- `Eval#program` qui sort un type `Observation.t maybe`

Toutes trois ont des types complexe mais utilisent des sorties similaire et doivent être modifiées similairement en plus de prendre en paramètre un type `(Typ.t * TPath.t)` elles doivent renvoyer un type `(Typ.t * TPath.t) list` ce qui complexifie énormément le code de tout le programme.

Tout ceci pour que dans un module *Tabular* on puisse parcourir le type et construire recursivement les données pour le fichier CSV.

Le problème c'est que lorsque qu'on détecte une donnée manquante on est en train d'évaluer seulement une partie de l'observation initiale et que l'on n'a pas d'information sur la manière dont celle-ci s'intègre à l'observation initiale. On ne peut donc pas calculer l'entièreté du chemin correspondant à la donnée manquante à partir de ce point. Propager les informations nécessaires à ce calcul nécessitait de modifier l'entièreté du code de l'évaluateur, une solution complexe et peu élégante.

Il y a été décidé au bout d'un peu moins de trois semaines de travail que c'était une mauvaise approche, et que même si ça voulait dire mettre à la poubelle trois semaines de travail il fallait recommencer à zéro.

III.2.2 Le retour à zéro, une nouvelle approche plus simple : le pré-processing

Avec une branche propre, la première étape a été de récupérer le code du module *Tabular*, sauf la fonction pour générer les données, les autres fonctions étaient bonnes.

Cette nouvelle approche se base sur le principe de *pré-processing* nous allons utiliser une fonction `junkize` de type `Typ.t -> Observation.t -> TPath.t -> Observation.t` pour générer les chemins comme expliqué plus bas.

Pour continuer nous allons remplacer à chaque fois que l'on a une donnée vide dans une observation une donnée de type `Freeform Text` (qui contiendra le tpath correspondant à la donnée manquante) où l'on va mettre le `TPath` (ce qui répond aux difficultées de la première approche on a plus besoin de calculer le TPath mais juste de le récupérer et de le traiter) ce qui permet que dans la fonction `Pattern#matchp` quand l'on gère le filtrage par motif du programme on regarde quand est-ce que l'on a une somme pour le *type* mais un freeform pour *l'observation* et dans ce cas là on génère une nouvelle donnée par la fonction `junk` de type `Typ.t -> string ->`

`Observation.t` (elle aussi qui va ajouter le TPath pour les enfants) qui va être propagée récursivement.

Nous allons ensuite paramétrer c'est à dire rajouter un paramètre à une fonction, nos fonctions importantes par une fonction appelée `callback` de type `Typ.t -> TPath.t -> unit` (prenant ainsi appuis sur notre langage de programmation fonctionnel)

Cette méthode à l'avantage d'être beaucoup plus légère en terme de code et de compréhension. On a pas besoin de modifier tout l'environnement de typage ou de propager des listes de TPath partout. Juste une fonction callback.

Nous allons utiliser `junkize` pour modifier récursivement une observation en une nouvelle observation avec les données `Freeform Tpath`, ainsi que `junk` à appeler dans `Pattern#matchp`, pour générer la nouvelle observation ce style nous permet d'éviter d'avoir à se traîner des TPath partout. Et sont utilisées après avoir appelé notre callback.

Maintenant le tout est de modifier le comportement de `Pattern#matchp`, quand on tombe sur un type `Sum` mais une observation `Freeform Tpath`. C'est là que l'on peut appeler notre callback ! Qui rappelons le n'a besoin que d'un type et d'un TPath pour être appelée et renvoie un simple `unit`, le *« void »* de OCaml. Mais on se heurte à un problème, on ne peut pas continuer l'éxecution récursivement vu que l'on a que un `Freeform` en observation. C'est là que la fonction **`junk`** est utile!

On construit alors une `observation` de type `Sum` avec des constructeurs à la même probabilité (exemple si il y en a deux alors les deux seront à 50% de chance), et qui au lieu de continuer récursivement en profondeur comme dans Liste 2, ne continue que sur un niveau avant d'avoir de nouveau des `Freeform` avec des `TPath` cette fois avancé d'un niveau, un exemple possible est si l'on avait le TPath `flower.shape` on aura alors `floyer.shape.bent` et `flower.shape.straight`.

On continue ensuite l'évaluation de la fonction `Pattern#matchp` sur cette *observation* ci pour lui permettre d'avoir quelque chose sur quoi continuer récursivement.

III.2.3 Les limites avec cette approche

Cette approche a quelques problèmes, notamment de synchronisation des `Typ` avec les `Observation` (les observations évalués n'avait pas le même type que le type évalué par la fonction `matchp` ce qui ne marchait donc pas) pour que l'on ai pas de problèmes. Un problème de synchronisation a posé quelques heures de debug.

Un autre problème rencontré est que la base de code possède plusieurs fonctions faisant des manipulations sur des `Sum` et des `Record`, cependant la fonction `junkize` remplace des `Sum` par des `Freeform` quand elles sont vides ce qui cause les fonctions pensées pour manipuler uniquement des `Sum` à devoir manipuler des données de type `Freeform`.

Une fois ces problèmes corrigés je suis tombée sur un autre cas. Une désynchronisation arrivait alors ce qui générait un cas non prévu à la fonction `Pattern#matchp` et alors il tombait sur le cas *wildcard* `_` qui est le cas « général » ou « tout le reste » et celui ci génère un `assert false` qui fait planter le programme, nous indiquant que le programmeur (ici, moi) a fait une erreur quelque part.

III.2.4 Correction de l'approche

Il a fallu faire plusieurs changement pour corriger l'approche. Déjà j'ai dû modifier toutes les fonctions où il y avait des `Sum` pour prendre en compte des `Freeform` et ensuite sous les conseils de mes responsables de stage j'ai changé ma fonction `junkize`. En effet le problème de désynchronisation venait du fait que j'avais artificiellement « gonflé » les listes pour les fusionner et pouvoir avancer.

Pour corriger ce problème j'ai utilisé un argument des *observations* pour appeler le bon constructeurs dans les *type* `Sum` afin d'avoir tout ce qu'il me fallait pour continuer récursivement. Pour les `Sum` et les `Record`.

Ensuite le fichier était généré mais pas correctement! En effet comme j'avais oublié d'effacer la table utilisée par la *callback* toutes les questions étaient reposées encore et encore ce qui faisait que j'avais un fichier `CSV` de près de 500 kilo-octets. Une fois la table effacée à chaque nouveau taxon le fichier ne fait plus que 5 kilo-octets, et cette fois tous les chemins sont bien des chemins manquant dans la base de donnée correspondante.

III.2.5 La fonction inverse

C'est bien d'avoir un fichier CSV que les botanistes peuvent remplir pour répondre plus facilement aux questions mais il faut pouvoir prendre ce fichier CSV et l'utiliser pour remplir les données correspondantes dans la collection.

À ce but nous allons faire une fonction `of_csv` qui charge depuis le fichier CSV et une fonction `apply` pour l'appliquer à la collection donnée.

Cette fonction parcours récursivement *l'observation* associée à la collection avec le type et chaque `TPath` associé à un taxon. Il remplace alors les données des cases vides par la valeur associé.

III.2.6 Conclusion des chemins manquant

J'ai sur-estimé ma capacité à avancer quand j'ai commencé à faire cette partie, je pensais en avoir pour deux ou trois semaines mais comme vu précédemment il m'a fallu 3 semaines pour *recommencer à zéro* ce sont des choses qui arrivent.

Sur la deuxième approche j'ai *sous-estimé* la quantité de bug auxquels j'allais faire face qui ont pris beaucoup de temps pour être corrigés et pouvoir avancer.

Mais j'ai réussi à faire jusqu'à la fonction inverse ce qui est la limite de ce qui a été demandé. Même si je n'ai pu faire que le début.

On verra dans la partie perspective ce que je compte faire avec le temps supplémentaire (2 semaines) qu'il me reste avant de devoir finir ce stage.

IV Bilan et perspectives

IV.1 Bilan

Le bilan de ce stage est très positif, non seulement une grande partie de ce qui a été demandé a été réalisé mais le code produit sera directement utilisé pour être utilisé soit par les chercheurs en informatique soit par les chercheurs en botanie, ce qui est rare et n'était pas le cas du stage de l'an dernier où le bilan était bon mais purement théorique.

Le framework de tests va être utilisé sous peu pour générer des tests de régression pour permettre de jouer avec le traducteur avec la garantie que l'on ne baisse pas les fonctionnalités de celui ci, et la détection de

données manquante va permettre d'économiser du temps de botaniste.

Bien qu'ayant fais du OCaml avant (en prépa) je ne connaissais pas nombre de techniques ayant été utilisées sur ce stage notamment les fameux *ppx* ainsi que les monades ou le fait de pouvoir faire une fonction du même nom pour tracer les appels. Ces morceaux de code et techniques me permettrons de devenir une meilleure développeuse en OCaml et de me perfectionner.

Les difficultées rencontrées m'ont permis de toucher un peu au métier de la recherche, d'apprendre ce que font les chercheurs, d'apprendre l'humilitée de devoir recommencer à zéro après trois semaines de travail *ce qui représente quand même la moitié de ce morceau de six semaines de mon stage*. Tout ceci me permet de voir quel est le métier de la recherche et si j'ai envie d'en faire mon métier ou non.

J'ai regretté ne pas avoir vu plus tôt de devoir recommencer plus tôt pour gagner du temps même si ce sont mes responsables de stage qui ont fait le choix de recommencer ce qui me met à l'aise avec le fait de ne pas l'avoir vu. Je regrette n'avoir pas pu être plus présente sur le centre et travailler plus tard pour faire plus cependant je suis satisfaite du travail que j'ai accomplis en six semaines et il m'en reste encore deux.

IV.2 Perspectives

Il me reste deux semaines dans ce stage je peux donc espérer pouvoir corriger mes bugs et finir la fonction inverse et ainsi donc terminer la partie sur les chemins manquant. Je ne pense pas pouvoir achever plus de travail dans ces deux dernières semaines, autre que faire des petites tâches et documenter du code. Deux semaines restent assez courtes.

Mais si je réussi à finir cette partie j'aurais contribué au projet et les botanistes pourrons alors plus facilement remplir les chemins les plus importants.

V Remerciements

Je tiens à remercier *M. Simon Castellan* et *Mme Aurore Alcolei* pour leur accueil, leurs conseils, leur patience, et leur jovialité durant ce stage.

Je voudrais aussi à remercier toute l'équipe *EPICURE* pour son accueil chaleureux au sein de leur équipe tout au long de ce stage et eux aussi de leurs conseils et discussions qui se sont avérés être indispensable aux resultats de ce stage.

Et plus généralement un remerciement au personnel - que ce soit de recherche ou non - du Centre Inria de l'Université de Rennes et de l'IRISA pour leur accueil et leurs services.

Bibliographie

[1] S. Castellan, J. Käfer, et E. Tannier, « Back to the trees: Identifying plants with Human Intelligence », in *Ninth Computing within Limits 2023*, juin 2023. [En ligne]. Disponible sur: https://hal.science/hal-04121511v1/document

[2] Back-To-The-Trees, « Code de Plantinator ». [En ligne]. Disponible sur: https://gitlab.inria.fr/back-to-the-trees/plantinator

[3] EPICURE, « EPICURE Website ». [En ligne]. Disponible sur: https://team.inria.fr/epicure