Rapport TP 1

Laurence Leblond
William Méroz-Moreau

18 octobre 2024

1 Problèmes rencontrés

1.1 Connaissance trop minime de Haskell

Haskell n'est pas un langage très intuitif à prime abord. Penser et déconstruire un problème par récursions semble être un défi de taille pour ceux qui n'y ont pas l'habitude. Mais à travers les différents problèmes rencontrés, notamment dans ce TP et les devoirs, en élargissant pas à pas notre sphère d'aisance, nous sommes arrivés à avoir une bonne prise en main de la logique du langage. Tel que mentionné dans le cours, il est possible (et souhaitable) de considérer un code comme une preuve mathématique, en établissant systématiquement le chemin des déductions qui partent du ou des types d'entrée jusqu'au type de sortie, et Haskell s'y prête grandement. En ayant en tête l'idée de preuve mathématique (notamment par induction), la pensée par récursion devient plus simple : on débute par le cas de base, puis par un cas n et $n+1$ qui représentent les autres cas de figures. Penser de telle manière est pratique pour bien comprendre le fonctionnement des fonctions et pour anticiper les erreurs. Nous avons donc essayé de suivre cette démarche logique et systématique dans notre TP, bien que ce ne soit pas un muscle à l'usage familier. En gardant ce train de penser, nous avons pu garder en tête tous les scénarios possibles qui devaient être couverts, afin d'éviter de rendre notre code, et de ce fait notre évaluateur, plus robuste.

1.2 Compréhension de la donnée

Pour bien comprendre l'énoncé du TP, nous avons décidé de le lire chacun de notre côté puis de se rencontrer pour résumer les différentes parties. En déconstruisant le travail de la sorte, nous avons mieux compris les différentes tâches que nous avions à faire. Il nous a d'abord été compliqué de comprendre les exemples de code Slip donnés dans l'énoncé, la syntaxe Lisp étant nouvelle pour nous. Il nous a également pris du temps à comprendre ce qui était demandé de nous.

1.3 Lire, trouver et comprendre les parties importantes du code fourni

Pour ce qui est du code donné, la longueur du fichier nous a d'abord parut inquiétante et nous avons surévalué la tâche à faire. Mais en ayant bien défini les sections à compléter grâce à notre synthèse de la donnée, nous avons simplement considéré de gros bouts du code comme des boites noires pour nous concentrer exclusivement sur $s2l$ et $eval$. [développer] Avoir travailler sur la construction d'évaluateurs dans les différents devoirs nous a permis de comprendre ce que nous devions faire dans $eval$ assez rapidement, mais nous avions quelques doutes sur la manière dont fonctionnait $s2l$ et comment était défini la structure des $sexp$ et des $lexp$. Nous avons débuté par lister les cas que nous devions effectuer. Pour $s2l$, il était trivial que nous avions à transformer les expressions $Snil$, $Ssym$, $Snum$ et $Snode$ en $Lnum$, $Lbool$, $Lvar$, $Ltest$, $Lfob$, $Lsend$, $Llet$ et $Lfix$. Puis pour $eval$, il fallait prendre ces derniers cas et les transformer en $Vnum$, $Vbool$, $Vbuiltin$ et $Vfob$. Avant de commencer à coder, nous voulions comprendre la logique des transformations (et particulièrement $s2l$). Les cas les plus simples étaient ceux $Snum$ et des $Ssym$ que nous comprenions assez bien, mais pour les autres, il nous semblait très difficile de trouver un point de départ. Que doit faire $s2l$ au juste ? Comment sera t-il possible d'interpréter un $sexp$ en $lexp$ pour des booléens, des fonctions, etc. ? Nous pensions déjà en termes sémantiques, sur la valeur des arguments, des opérations, etc. Nous faisions fausse route et cela nous a conduit à considérer le travail comme beaucoup plus difficile qu'il ne l'était. Lorsque nous avions bien intériorisé le fait que $s2l$ manipule tout simplement des arbres syntaxique, le problème nous est apparu plus simple. Nous comprenions qu'il n'était que question de quelle disposition de symboles $Sexp$ donne telle disposition de symboles $Lexp$.

1.4 Compléter le code fourni

Ainsi le problème bien opérationnalisé, nous avons simplement commencer par les cas de figure qui nous semblaient simples, à savoir, les booléens, qui sont syntaxiquement simples (avec deux choix : "$true$" ou "$false$") puis ceux qui débutaient par un symbole particulier ("if", "fob", "let", "fix"). if nous a semblé relativement trivial ; nous avons trois expressions, une pour l'énoncé conditionnel, une pour la condition "true" et une pour la condition "false" et nous retournons un ltest avec chacune des expressions transformées en $Lexp$. Pour let, la transformation nous semblait aussi assez simple ; on prend une assignation et un corps d'expression, puis nous retournons tout simplement un $Llet$ avec la variable assignée, la transformation de la valeur assignée et la transformation du corps de l'expression. La logique nous semblait similaire pour $Lfix$ mais de manière récursive sur un ensemble indéterminé d'assignations. Notre approche était de d'établir la logique des fonctions $s2l$ puis celle de $eval$ avant de tester. Notre méthode nous semblait bonne, mais nous nous sommes rendu compte en cours de route de son inéficacité. La plus grande difficulté, par contre, fut l'implémentation des paternes nécessaires à l'interprétation des fonctionna-

lités fob, fix et les appels de fonctions ($Lsend$). Il nous a fallu du temps avant de comprendre la syntaxe même de ces fonctionnalité en code Slip. Pour les fob, nous n'avions pas compris à quel point il pouvait être difficile d'ajouter les bonnes valeurs à l'environnement de la fonction en plus de pouvoir retrouver ces valeurs de l'environnement quand vient de le temps d'exécuter la fonction. Pour les $Lsend$, le plus dur a été de comprendre comme extraire les arguments et les liés aux bonne variables (paramètres) de la fonction durant l'appel d'une fonction définie par l'utilisateur. Les appels de fonctoins prédéfinies étaient beaucoup plus simples. Le développement des fonctionnalités reliées au mot clé fix ont été le plus compliquées. Il fallait tout dabord comprendre le contexte dans lequel fix est utilisé, et pour quelle raison. Ensuite, nous avons dû essayer de comprendre tous les cas possibles et définir des règles où fix est sensé retourner une erreur. Seulement ensuite nous avons pu l'implémenter. Le code des fonctions $s2l$ et $eval$ pour $Lfix$ n'est pas très intuitif, et il nous a prit un grand nombre d'heures avant de comprendre que fix devrait faire l'utilisation de son propre environnement, en plus de devoir faire la distinction entre ce qui est une assignation d'une nouvelle variable et ce qui est une déclaration d'une nouvelle fonction. $Lfix$ est vraiment la fonctionnalité où tout le reste du code est mit en oeuvre pour complété la tâche demandée.

2 Choix

2.1 Garder les implémentations de fonctions le plus court possible

Nous avons tenté de garder les implémentations des différents paternes des fonctions $s2l$ et $eval$ le plus court possible afin de ne pas surcharger l'évaluateur avec trop de logique qui pourrait au final créer plus de bogues, et rendre ceux-ci plus difficile à identifier. Le code Slip ayant une syntaxe et des fonctionnalités assez simple, il est possible de déléguer beaucoup des tâches de l'analyse syntaxique et de l'évaluation aux mêmes fonctions pour beaucoup de scénarios différents. Par exemple, le code des fonctions $s2l$ et $eval$ pour les $Lfob/Vfob$ et $Llet$ peuvent se résumer à quelques lignes. Par contre, cette façon de travailler nous à causer du fil à retorde lors qu'est venu le temps d'implémenter la fonctionnalité des $Lfix$. En effet, puisque nos autres fonctions étaient autant dépourvues de fonctionnalité, il nous a fallu écrire beaucoup de code afin d'extraire les assignations et les noms d'arguments afin de les ajouter dans leur environnement. Ces traitements auraient peut-être pu être ajoutés dans les fonctions $s2l$ et $eval$ des $Lfob$ directement afin d'éviter de sur complexifier le code et mélanger les rôles des différents paternes.

2.2 Ne pas respecter toutes les recommandations faites dans l'énoncé

Nous avons commencé le TP en essayant de travailler le plus possible en binôme afin de s'entre-aider à bien comprendre la logique que nous devions suivre pour compléter le code. Au début du travail, ces heures passées à travailler ensemble étaient utiles, mais, nous nous sommes rendus compte qu'il nous était plus facile de se séparer l'implémentation en terme de fonctionnalité afin de se laisser à chacun la possibilité de se plonger dans le code et réfléchir sur son propre temps. Nous avons réaliser par contre que beaucoup de fonctionnalité dépendent des autres (on peut seulement bien tester les $Lfob$ et $Vfob$ que lorsque la fonctionnalité du $Lsend$ fonctionne bien, par exemple). Une autre recomendation était de commencer par implémenter l'évaluation du sucre syntaxique pour la création d'une nouvelle fonction anonyme. Après une multitude de tentatives d'implémenter cette fonctionnalité, nous avons décider de procéder avec l'implémentation de tout le code et de ensuite de soucier de l'implémentation du sucre syntaxique. Avec le recul, nous avons réaliser que ce fut une erreur puisque l'implémentation du sucre syntaxique après avoir écrit tout le reste du code est beaucoup plus difficile. Plus précisément, il est difficile de faire la distinction entre un paterne de $Lsend$ et un paterne relié au sucre syntaxique. Les 2 expressions sont très similaires, avec des $Snode$ imbriqués de façons similaires, et il aurait fallu modifier plusieurs parties du code pour le faire fonctionner correctement. Malgré tout nos efforts, nous ne sommes pas parvenus à le faire marcher correctement sans impacter le reste de la logique et créer des éffets de bord non désirés. Si nous avions suivi la recomendation de l'énoncé en commençant pas le sucre syntaxique. Nous n'aurions probablement pas eu ce problème.

3 Options rejetées

3.1 Inclure les appels de fonctions anonyme dans $Lfob$

Durant l'implémentation de la fonctionnalité d'appel d'une fonction anonyme ($Lfob$), nous avons commencé par essayer d'implémenter cette fonctionnalité directement dans le code pour l'interprétation des $Lfob$. Nous n'avions pas compris que puisque la syntaxe Slip $((fob(x)(+x1)2)$, par exemple, constituait en réalité une déclaration et un appel de fonction en même temps, il était plus sage d'implémenté cette fonctionnalité dans le code pour les $Lsend$. Après plusieurs jous à essayer de l'implémenter dans les $Lfob$, nous avons rejeté cette possibilité.

4 Surprises

4.1 Gestion des environnements

Nous ne nous attentions pas à ce que la gestion des environnements soit autant complexe. Slip étant un langage avec une portée statique, nous nous attentions à pouvoir simplement ajouter les données nécéssaires à l'environnement de base $env0$ pour ensuite avoir accès à toute variable et sa valeur peut importe l'endroit de l'appel de nos fonctions. Non seulement cette façon de penser démontre une mauvaise compréhension du concept de portée statique, elle est également impossible venant du fait que l'envrionnement $env0$ est immutable dans le code Haskell. Nous avons dû, après avoir réalisé notre erreur, programmer les $Lfob$ et, plus difficilement, les $Lfix$ afin qu'une « sauvegarde » de l'environnement dans lequel la fonction a été crée soit faite et soit garder en mémoire dans les $Lfob$. L'appel des fonctions avec $Lsend$ s'est fait sans trop de soucis une fois que nous avions compris le processus à suivre.

4.2 Importance du code avec faible couplage

Nous nous sommes vites rendu compte de l'importance d'écrire du code robuste qui pouvait être réutilisé pour chaque paternes que nous avons fais pour les fonctions $s2l$ et $eval$. En effet, dépendamment du code Slip que nous essayons d'interpréter, il est possible que toutes les options de $s2l$ ou $eval$ soient appelées pour accomplir une tâche commune. Beaucoup de fonctions vont dépendre des sorties d'autres fonctions. Si le code est bien fait, ces opérations devraient se faire sans réel interventions de notre part et tout devrait se lier de façon cohésive.