Travail pratique #1

IFT-2035

11 octobre 2023

1 Survol

Ce TP vise à améliorer la compréhension des langages fonctionnels en utilisant un langage de programmation fonctionnel (Haskell) et en écrivant une partie d'un interpréteur d'un langage de programmation fonctionnel (en l'occurence une sorte de Lisp). Les étapes de ce travail sont les suivantes :

1. Parfaire sa connaissance de Haskell.
2. Lire et comprendre cette donnée. Cela prendra probablement une partie importante du temps total.
3. Lire, trouver, et comprendre les parties importantes du code fourni.
4. Compléter le code fourni.
5. Écrire un rapport. Il doit décrire **votre** expérience pendant les points précédents : problèmes rencontrés, surprises, choix que vous avez dû faire, options que vous avez sciemment rejetées, etc... Le rapport ne doit pas excéder 5 pages.

Ce travail est à faire en groupes de deux. Le rapport, au format LaTeX exclusivement (compilable sur `ens.iro`) et le code sont à remettre par remise électronique avant la date indiquée. Aucun retard ne sera accepté. Indiquez clairement votre nom au début de chaque fichier.

Ceux qui veulent faire ce travail seul(e)s doivent d'abord en obtenir l'autorisation, et l'évaluation de leur travail n'en tiendra pas compte. Des groupes de 3 ou plus sont **exclus**.

$e ::= n$	Un entier signé en décimal
$\mid x$	Une variable
$\mid (\lambda\ x\ e)$	Une fonction avec un argument
$\mid (e_0\ e_1\ ...\ e_n)$	Un appel de fonction (*curried*)
$\mid (\mathsf{ref}\,!\ e)$	Construction d'une *ref-cell*
$\mid (\mathsf{get}\,!\ e)$	Chercher la valeur de la *ref-cell* e
$\mid (\mathsf{set}\,!\ e_1\ e_2)$	Changer la valeur de la *ref-cell* e_1
$\mid + \mid - \mid * \mid /$	Opérations arithmétiques prédéfinies
$\mid < \mid > \mid = \mid <= \mid >=$	Opérations booléennes sur les entiers
$\mid (\mathsf{if}\ e_1\ e_2\ e_3)$	Si e_1 alors e_2 sinon e_3
$\mid (\mathsf{let}\ x\ e_1\ e_2)$	Déclaration locale simple
$\mid (\mathsf{letrec}\ ((x_1\ e_1)\ ...\ (x_n\ e_n))\ e)$	Déclarations locales récursives

FIGURE 1 – Syntaxe de Slip

2 Slip : Une sorte de Lisp

Vous allez travailler sur l'implantation d'un langage fonctionnel dont la syntaxe est inspirée du langage Lisp. La syntaxe de ce langage est décrite à la Figure 1. À remarquer que comme toujours avec la syntaxe de style Lisp, les parenthèses sont *significatives*.

La forme let est utilisée pour donner un nom à une définition non-récursive locale. La forme letrec est une variante plus complexe qui autorise plusieurs définitions locales simultanées qui peuvent de plus être récursives, y compris mutuellement récursives, comme dans le let de Haskell.

2.1 Sémantique dynamique

Slip, comme Lisp, est un langage typé dynamiquement, c'est à dire que ses variables peuvent contenir des valeurs de n'importe quel type. Il n'y a donc pas de sémantique statique (règles de typage).

Les valeurs manipulées à l'exécution par notre langage sont les entiers, les fonctions, les booléens, et les références qui pointent vers des *ref-cell*, c'est à dire des cellules mémoire dont on peut modifier le contenu (le reste des valeurs manipulées par Slip sont immuables).

On construit une *ref-cell* avec ref ! qui prend comme argument la valeur initiale de cette cellule, alloue cette cellule quelque part en mémoire, et renvoie une référence (son *adresse*). On peut ensuite aller chercher la valeur actuelle de cette cellule avec get ! et changer la valeur avec set !.

Ainsi la fonction suivante :

```
(λ p (set! p (+ 1 (get! p))))
```

est une fonction qui incrémente une cellule en y ajoutant 1.

Les règles d'évaluation fondamentales sont les suivantes :

$$\begin{array}{rcl}
((\lambda\ x\ e)\ v) & \rightsquigarrow & e[v/x] \\
(\mathsf{let}\ x\ v\ e) & \rightsquigarrow & e[v/x] \\
(\mathsf{letrec}\ ((x_1\ v_1)\ ...\ (x_n\ v_n))\ e) & \rightsquigarrow & e[v_1, ..., v_n/x_1, ..., x_n] \\
(e_0\ e_1\ ...\ e_n) & \rightsquigarrow & ((e_0\ e_1)\ ...\ e_n) \\
(e_0) & \rightsquigarrow & e_0
\end{array}$$

où la notation $e[v/x]$ représente l'expression e dans un environnement où la variable x prend la valeur v. L'usage de v dans les règles ci-dessus indique qu'il s'agit bien d'une valeur plutôt que d'une expression non encore évaluée. Par exemple le v dans la première règle indique que lors d'un appel de fonction, l'argument doit être évalué avant d'entrer dans le corps de la fonction, i.e. on utilise l'appel par valeur.

En plus des deux règles β ci-dessus, les différentes primitives arithmétiques se comportent comme on s'y attend :

$$\begin{array}{rcl}
(+\ n_1\ n_2) & \rightsquigarrow & n_1 + n_2 \\
(-\ n_1\ n_2) & \rightsquigarrow & n_1 - n_2 \\
(*\ n_1\ n_2) & \rightsquigarrow & n_1 \times n_2 \\
... & \rightsquigarrow & ...
\end{array}$$

Donc il s'agit d'une variante du λ-calcul, sans grande surprise. La portée est statique et l'ordre d'évaluation est *par valeur*.

Les règles de réductions ci-dessous ne tiennent cependant pas compte de la présence des opérations impures ref !, get !, et set !, qui obligent à utiliser des réductions plus complexes où on ne considère pas que l'expression à réduire mais aussi l'état de la mémoire (aussi appelé parfois le *tas*).

Plus précisément les règles ci-dessus ont la forme ee' mais devraient en réalité être de la forme $(M; e)(M; e')$. Ainsi les règles impures peuvent se définir comme suit :

$$\begin{array}{rcll}
(M; (\mathsf{ref}\ !\ v)) & \rightsquigarrow & (M, \ell \mapsto v; \ell) & \ell \text{ est une nouvelle adresse} \\
(M; (\mathsf{get}\ !\ \ell)) & \rightsquigarrow & (M; v) & M(\ell) = v \\
(M; (\mathsf{set}\ !\ \ell\ v)) & \rightsquigarrow & (M'; v) & M' = M, \ell \mapsto v
\end{array}$$

Le code Haskell reste pur et va donc suivre ces règles de sémantiques de manière assez directe : la fonction eval va recevoir un argument qui représente ce M qui sera modifié de manière pure.

3 Implantation

L'implantation du langage fonctionne en plusieurs phases :

1. Une première phase d'analyse lexicale et syntaxique transforme le code source en une représentation décrite ci-dessous, appelée *Sexp* dans le code. Ce n'est pas encore tout à fait un arbre de syntaxe abstraite (cela s'apparente en fait à XML).

2. Une deuxième phase, appelée *s2l*, termine l'analyse syntaxique et commence la compilation, en transformant cet arbre en un vrai arbre de syntaxe abstraite dans la représentation appelée *Lexp* dans le code. Dans un sens, cette phase commence déjà la compilation vu que le langage *Lexp* n'est pas identique à notre langage source.
3. Finalement, une fonction *eval* procède à l'évaluation de l'expression par interprétation.

Une partie de l'implantation est déjà fournie : la première ainsi que divers morceaux des autres. Votre travail consistera à compléter les trous.

3.1 Analyse lexicale et syntaxique : *Sexp*

L'analyse lexicale et syntaxique est déjà implantée pour vous. Elle est plus permissive que nécessaire et accepte n'importe quelle expression de la forme suivante :

$$e ::= n \mid x \mid \text{`('} \{ e \} \text{`)'}$$

n est un entier signé en décimal.
Il est représenté dans l'arbre en Haskell par : Snum n.

x est un symbole qui peut être composé d'un nombre quelconque de caractères alphanumériques et/ou de ponctuation. Par exemple '+' est un symbole, '<=' est un symbole, 'voiture' est un symbole, et 'a+b' est aussi un symbole. Dans l'arbre en Haskell, un symbole est représenté par : Ssym x.

'(' { e } ')' est une liste d'expressions. Dans l'arbre en Haskell, les listes d'expressions sont représentées par des listes simplement chaînées constituées de paires Scons *left right* et du marqueur de fin Snil. *left* est le premier élément de la liste et *right* est le reste de la liste.

Par exemple l'analyseur syntaxique transforme l'expression `(+ 2 3)` dans l'arbre suivant en Haskell :

```
Snode (Ssym "+")
      [Snum 2, Snum 3]
```

L'analyseur lexical considère qu'un caractère ';' commence un commentaire, qui se termine à la fin de la ligne.

3.2 La représentation intermédiaire *Lexp*

Cette représentation intermédiaire est une sorte d'arbre de syntaxe abstraite. Dans cette représentation, +, -, ... sont simplement des variables prédéfinies.

3.3 L'environnement d'exécution

Le code fourni défini aussi l'environnement initial d'exécution, qui contient les fonctions prédéfinies du langage telles que l'addition, la soustraction, etc. Il est défini comme une table qui associe à chaque identificateur prédéfini la valeur (de type *Value*) associée.

3.4 Évaluation

L'évaluateur utilise l'environnement initial pour réduire une expression (de type *Lexp*) à une valeur (de type *Value*). Il prend de plus un argument qui représente l'état de la mémoire M et qui est renvoyé (possiblement modifié) avec la valeur résultant de l'évaluation de l'expression.

4 Cadeaux

Comme mentionné, l'analyseur lexical et l'analyseur syntaxique sont déjà fournis.

Voilà ci-dessous un exemple de session interactive sur une machine GNU/Linux, avec le code fourni :

```
% ghci slip.hs
GHCi, version 9.0.2: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main             ( slip.hs, interpreted )

slip.hs:221:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘hinsert’:
        Patterns not matched:
            Hempty _ (Vnum _)
            Hempty _ (Vbool _)
            Hempty _ (Vref _)
            Hempty _ (Vfun _)
            ...
    |
221 | hinsert _ p _ | p < 0 = error "hinsert sur une adresse négative"
    | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

slip.hs:295:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘eval’:
        Patterns not matched:
            (_, _) [] (Lid _)
            (_, _) [] (Labs _ _)
            (_, _) [] (Lfuncall _ _)
            (_, _) [] (Lmkref _)
            ...
    |
295 | eval s _env (Llit n) = (s, Vnum n)
    | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Ok, one module loaded.
ghci> run "exemples.slip"
[2,*** Exception: slip.hs:295:1-34: Non-exhaustive patterns in function eval
```

```
ghci>
```

Les avertissements et l'exception levée sont dus au fait que le code a besoin de vos soins. Le code que vous soumettez ne devrait pas souffrir de tels avertissements, et l'appel à `run` devrait renvoyer la liste des valeurs décrites en commentaires dans le fichier.

5 Recommendations

Je recommande de le faire "en largeur" plutôt qu'en profondeur : compléter les fonctions peu à peu, pendant que vous avancez dans `exemples.slip` plutôt que d'essayer de compléter tout *s2l* avant de commencer à attaquer la suite. Ceci dit, libre à vous de choisir l'ordre qui vous plaît.

De même je vous recommande fortement de travailler en binôme (*pair programming*) plutôt que de vous diviser le travail, vu que la difficulté est plus dans la compréhension que dans la quantité de travail.

Le code contient des indications des endroits que vous devez modifiez. Généralement cela signifie qu'il ne devrait pas être nécessaire de faire d'autres modifications, sauf ajouter des fonctions auxiliaires. Ceci dit, vous pouvez aussi modifier le reste du code, si vous le voulez, mais il faudra alors justifier ces modifications dans votre rapport en expliquant pourquoi cela vous a semblé nécessaire.

Vous devez aussi fournir un fichier de tests `tests.slip`, similaire à `exemples.slip`, mais qui contient au moins 5 tests que *vous* avez écrits (avec en commentaire la valeur de retour que vous pensez devrait être renvoyée).

5.1 Remise

Pour la remise, vous devez remettre trois fichiers (`slip.hs`, `tests.slip`, et `rapport.tex`) par la page Moodle (aussi nommé StudiUM) du cours. Assurez-vous que le rapport compile correctement sur `ens.iro` (auquel vous pouvez vous connecter par SSH).

6 Détails

- La note sera divisée comme suit : 25% pour le rapport, 60% pour le code (réparti entre *s2l*, hinsert, et *eval*), et 15% pour les tests.
- Tout usage de matériel (code ou texte) emprunté à quelqu'un d'autre (trouvé sur le web, ...) doit être dûment mentionné, sans quoi cela sera considéré comme du plagiat.
- Le code ne doit en aucun cas dépasser 80 colonnes.
- Vérifiez la page web du cours, pour d'éventuels errata, et d'autres indications supplémentaires.

— La note est basée d'une part sur des tests automatiques, d'autre part sur la lecture du code, ainsi que sur le rapport. Le critère le plus important, et que votre code doit se comporter de manière correcte. Ensuite, vient la qualité du code : plus c'est simple, mieux c'est. S'il y a beaucoup de commentaires, c'est généralement un symptôme que le code n'est pas clair ; mais bien sûr, sans commentaires le code (même simple) est souvent incompréhensible. L'efficacité de votre code est sans importance, sauf si votre code utilise un algorithme vraiment particulièrement ridiculement inefficace.