Travail pratique #1

IFT-2035

14 mai 2022

¡¡ Dû le 3 juin à 23h59 ! !

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'occurrence 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 2 étudiants. 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.

Si un étudiant préfère travailler seul, libre à lui, mais l'évaluation de son travail n'en tiendra pas compte. Des groupes de 3 ou plus sont **exclus**.

τ ::=	Int	Type des nombres entiers
	$\mid (\tau_1 \ ... \ \tau_n \to \tau)$	Type d'une fonction
	$\mid$ (List τ)	Type d'une liste
e ::=	n	Un entier signé en décimal
	$\mid x$	Référence à une variable
	$\mid (e_0 \ e_1 \ ... \ e_n)$	Un appel de fonction (*curried*)
	$\mid$ (let $((x_1 \ e_1) \ ... \ (x_n \ e_n)) \ e$)	Ajout de variables locales
	$\mid$ (letfn $f \ ((x_1 \ \tau_1) \ ... \ (x_n \ \tau_n)) \ \tau \ e_1 \ e_2$)	Définition de fonction
	$\mid (: e \ \tau)$	Annotation de type
	$\mid + \mid - \mid * \mid /$	Opérations arithmétiques
	$\mid$ (cons $e_1 \ e_2$) $\mid$ (nil τ) $\mid$ (list $\tau \ e_1 \ ... \ e_n$)	Construction de listes
	$\mid$ (case $e \ b_1 \ ... \ b_n$)	Filtrage sur les listes
b ::=	(nil e) $\mid$ ((cons $x_1 \ x_2$) e)	Branche de filtrage

FIGURE 1 – Syntaxe de Psil

2 Psil : 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. Mais contrairement à la tradition de Lisp, le type des fonctions $(\tau_1 \to \tau_2)$ utilise une syntaxe infixe plutôt que préfixe.

La fonction prédéfinie cons construit un élément de liste (comme le (:) de Haskell) ; (nil τ) est une constante prédéfinie utilisée pour représenter la liste vide (comme [] en Haskell, sauf qu'elle vient avec une annotation de type pour ne pas avoir besoin de l'inférer).

La forme let est utilisée pour donner des noms à des définitions locale. Elle n'autorise pas la récursion. Exemple :

```
(let ((x 2)          ⇝*  5
     (y 3))
  (+ x y))
```

Contrairement à Haskell, le langage oblige à nommer les fonctions, que l'on peut définir avec (letfn *f args* τ *body e*) où τ est le type de la valeur de retour, *body* est le corps de la fonction et "*e*" est le reste du calcul une fois que *f* est défini :

```
(letfn sum ((xs (List Int))) Int
  (case xs
    (nil 0)                          ⇝*  11
    ((cons x xs) (+ x (sum xs))))
 (sum (list Int 5 6)))
```

2.1 Sucre syntaxique

Les fonctions n'ont en réalité qu'un seul argument : la syntaxe offre la possibilité de déclarer et de passer plusieurs arguments, mais ces arguments sont passés par *currying*. Dans le même ordre d'idée, le constructeur list est une forme simplifiée équivalente à une combinaison de cons et de nil. Plus précisément, les équivalences suivantes sont vraies pour les expressions :

$$\begin{array}{rcl}
(e_0\ e_1\ e_2\ ...\ e_n) & \iff & (..((e_0\ e_1)\ e_2)\ ...\ e_n) \\
(\tau_1\ ...\ \tau_n \to \tau) & \iff & (\tau_1 \to ...(\tau_n \to \tau)..) \\
(\mathsf{list}\ \tau\ e_1\ ...\ e_n) & \iff & (\mathsf{cons}\ e_1 ...(\mathsf{cons}\ e_n\ (\mathsf{nil}\ \tau))..) \\
(\mathsf{let}\ ((x_1\ e_1)\ ...\ (x_n\ e_n))\ e) & \iff & (\mathsf{let}\ ((x_1\ e_1))\ ...\ (\mathsf{let}\ ((x_n\ e_n))\ e)..)
\end{array}$$

Votre première tâche sera d'écrire une fonction *s2l* qui va "éliminer" le sucre syntaxique, c'est à dire faire l'expansion des formes de gauche (présumément plus pratiques pour le programmeur) dans leur équivalent de droite, de manière à réduire le nombre de cas différents à gérer dans le reste de l'implantation du langage. Cette fonction va aussi transformer le code dans un format plus facile à manipuler par la suite.

On pourrait aussi définir un sucre syntaxique similaire pour la définition de fonction :

$$\begin{array}{l}
(\mathsf{letfn}\ f\ ((x_1\ \tau_1)\ (x_2\ \tau_2))\ \tau\ e_1\ e_2) \\
\iff \quad (\mathsf{letfn}\ f\ ((x_1\ \tau_1))\ (\tau_2\ \to\ \tau)\ (\mathsf{letfn}\ f'\ ((x_2\ \tau_2))\ \tau\ e_1\ f')\ e_2)
\end{array}$$

sauf que ça nécessite d'inventer un nom f' alors on a préférer garder la forme plus complexe dans un premier temps.

2.2 Sémantique statique

Une des différences les plus notoires entre Lisp et Psil est que Psil est typé statiquement. Les règles de typage (voir Figure 2) utilisent le jugement $\Gamma \vdash e : \tau$ qui dit que l'expression e, est typée correctement et a type τ. Dans ces règles, Γ représente le contexte de typage, c'est à dire qu'il contient le type de toutes les variables qui existent là où e est placé.

Il manque les règles de typage des opérations arithmétiques, car elles sont généralement traitées simplement comme des "variables prédéfinies" qui sont donc incluse dans le contexte Γ initial.

La deuxième partie du travail est d'implanter la vérification de types, donc de transformer ces règles en un morceau de code Haskell. Un détail important pour cela est que le but fondamental de la vérification de types n'est pas de trouver le type d'une expression mais plutôt de trouver d'éventuelles erreurs de typage, donc il est important de tout vérifier.

2.3 Sémantique dynamique

Les valeurs manipulées à l'exécution par notre langage sont les entiers, les fonctions, et les listes (dénotées [], et $[v_1\ .\ v_2]$). De plus la notation de listes est

$$\frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\ : e\ \tau) : \tau}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1 \to \tau_2) \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2} \qquad \frac{}{\Gamma \vdash (\mathsf{nil}\ \tau) : (\mathsf{List}\ \tau)}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : (\mathsf{List}\ \tau)}{\Gamma \vdash (\mathsf{cons}\ e_1\ e_2) : (\mathsf{List}\ \tau)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash (\mathsf{let}\ ((x_1\ e_1))\ e_2) : \tau_2}$$

$$\frac{\Gamma' = \Gamma, f{:}(\tau_1\ ...\ \tau_n\ \to\ \tau_r) \quad \Gamma', x_1{:}\tau_1, ..., x_n{:}\tau_n \vdash e_1 : \tau_r \quad \Gamma' \vdash e_2 : \tau}{\Gamma \vdash (\mathsf{letfn}\ f\ ((x_1\ \tau_1)\ ...\ (x_n\ \tau_n))\ \tau_r\ e_1\ e_2) : \tau}$$

$$\frac{\Gamma \vdash e : (\mathsf{List}\ \tau_1) \quad \Gamma \vdash e_n : \tau_2 \quad \Gamma, x{:}\tau_1, xs{:}(\mathsf{List}\ \tau_1) \vdash e_c : \tau_2}{\Gamma \vdash (\mathsf{case}\ e\ (\mathsf{nil}\ e_n)\ ((\mathsf{cons}\ x\ xs)\ e_c)) : \tau_2}$$

FIGURE 2 – Règles de typage

étendue de sorte que $[v_1\ .\ [v_2\ .\ [v_3\ .\ v_4]]]$ se note $[v_1\ v_2\ v_3\ .\ v_4]$ et qu'un ".[]" final peut s'éliminer. Donc $[v_1\ v_2]$ est équivalent à $[v_1\ .\ [v_2\ .\ []]]$.

La règle d'évaluation du let est la suivante :

$$(\mathsf{let}\ ((x_1\ v_1))\ e) \quad \rightsquigarrow \quad e[v_1/x_1]$$

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 la règle ci-dessus indique qu'il s'agit bien d'une valeur plutôt que d'une expression non encore évaluée, i.e. on utilise l'appel par valeur. En plus de la règle ci-dessus, les différentes primitives se comportent comme suit :

$$\begin{aligned}
(+\ n_1\ n_2) &\rightsquigarrow n_1 + n_2 \\
(-\ n_1\ n_2) &\rightsquigarrow n_1 - n_2 \\
(*\ n_1\ n_2) &\rightsquigarrow n_1 \times n_2 \\
(/\ n_1\ n_2) &\rightsquigarrow n_1 \div n_2 \\
(\mathsf{nil}\ \tau) &\rightsquigarrow [] \\
(\mathsf{cons}\ v_1\ v_2) &\rightsquigarrow [v_1\ .\ v_2] \\
(\mathsf{case}\ []\ ...(\mathsf{nil}\ e_n)...) &\rightsquigarrow e_n \\
(\mathsf{case}\ [v_1\ .\ v_2]\ ...((\mathsf{cons}\ x\ xs)\ e_c)...) &\rightsquigarrow e_c[v_1, v_2/x, xs]
\end{aligned}$$

Donc il s'agit d'une variante du λ-calcul, sans grande surprise. La portée est lexicale et l'ordre d'évaluation est présumé être "par valeur", mais vu que le langage est pur, la différence n'est pas très importante pour ce travail.

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 S (ou *Sexp* pour "syntactic expression") dans le code. C'est une sorte de proto-ASA (arbre de syntaxe abstraite). Cette partie vous est offerte.
2. Une deuxieme 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 appellée L (ou *Lexp* pour "lambda expression") dans le code. Comme mentionné, cette phase commence déjà la compilation vu que le langage *Lexp* n'est pas identique à notre langage source. En plus de terminer l'analyse syntaxique, cette phase élimine le sucre syntaxique (i.e. applique les règles de la forme ... $\Longleftrightarrow$ ...), et doit faire quelques ajustements supplémentaire à `case`.
3. Une troisième phase, appelée *check*, vérifie que le code est correctement typé et en trouve son type.
4. Finalement, une fonction *eval* procède à l'évaluation de l'expression par interprétation.

Votre travail consiste à compléter *s2l*, *check*, et *eval*.

3.1 Analyse lexicale et syntaxique : *Sexp*

L'analyse lexicale et (une première partie de l'analyse) 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 début `Snil`. *right* est le dernier élément de la liste et *left* est le reste de la liste (i.e. ce qui le précède).

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

```
Scons (Scons (Scons Snil
                    (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, et le sucre syntaxique n'est plus disponible, donc par exemple les appels de fonctions ne prennent plus qu'un seul argument et la forme let ne peut définir qu'une variable.

Elle est définie par le type :

```
data Lexp = Lnum Int
          | Lvar Var
          | Lannot Lexp Ltype
          | Linvoke Lexp Lexp
          | Lnil Ltype
          | Lcons Lexp Lexp
          | Lcase Lexp Lexp Var Var Lexp
          | Llet Var Lexp Lexp
          | Lletrec Var [(Var, Ltype)] Ltype Lexp Lexp
          deriving (Show, Eq)
```

Le Linvoke est le nœud qui représente un appel de fonction. Lcase $e\ e_n\ x\ xs\ e_c$ correspond à l'expression conditionnelle (case e (nil e_n) ((cons x xs) e_c)). Le Lletrec correspond aux expressions letfn de Psil, mais on l'appelle ici "letrec" pour rappeller que cela permet les définitions *réc*ursives.

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.

4 Cadeaux

Comme mentionné, l'analyseur lexical et l'analyseur syntaxique sont déjà fournis. Dans le fichier `Psil.hs`, vous trouverez les déclarations suivantes :

Sexp est le type des arbres, il défini les différents noeuds qui peuvent y apparaître.

readSexp est la fonction d'analyse syntaxique.

showSexp est un pretty-printer qui imprime une expression sous sa forme "originale".

Lexp est le type de la représentation intermédiaire du même nom.

s2l est la fonction qui transforme une expression de type *Sexp* en *Lexp*.

check est la fonction qui infère (et vérifie) le type d'une expression.

tenv0 est l'environnement de typage initial.

venv0 est l'environnement d'évaluation initial.

Value est le type du résultat de l'évaluation d'une expression.

eval est la fonction d'évaluation qui transforme une expression de type *Lexp* en une valeur de type *Value*.

run est la fonction principale qui lie le tout ; elle prend un nom de fichier et applique *evalSexp* sur toutes les expressions trouvées dans ce fichier.

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

```
% ghci
GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
Prelude> :load "psil.hs"
[1 of 1] Compiling Main             ( psil.hs, interpreted )

psil.hs:229:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘check’:
        Patterns not matched:
            _ (Lannot _ _)
            _ (Linvoke _ _)
            _ (Lnil _)
            _ (Lcons _ _)
            ...
    |
229 | check _tenv (Lnum _) = Lint
    | ^^^^^^^^^^^^^^^^^^^^^^^^^^...

psil.hs:285:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘eval’:
        Patterns not matched:
            _ (Linvoke _ _)
            _ (Lnil _)
            _ (Lcons _ _)
            _ (Lcase _ _ _ _ _)
            ...
    |
285 | eval _env (Lnum n) = Vnum n
    | ^^^^^^^^^^^^^^^^^^^^^^^^^^...
Ok, one module loaded.
*Main> run "exemples.psil"
  2 : Lint
  <function> : Lfun Lint (Lfun Lint Lint)
  *** Exception: Malformed Psil: (+ 2)
CallStack (from HasCallStack):
  error, called at psil.hs:215:10 in main:Main
*Main>
```

Lorsque votre travail sera fini, il ne devrait plus y avoir d'avertissements, et `run` devrait renvoyer plus de valeurs que juste les deux ci-dessus et terminer sans erreurs.

5 À faire

Vous allez devoir compléter l'implantation de ce langage, c'est à dire compléter *s2l*, *check*, et *eval*. Je recommande de le faire "en largeur" plutôt qu'en profondeur : compléter les fonctions peu à peu, pendant que vous avancez dans `exemples.psil` 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 définitions auxiliaires. 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.psil`, qui, tout comme `exemples.psil` devrait non seulement contenir du code Psil mais aussi indiquer les valeurs et types qui lui corresponde (à votre avis). Il doit contenir au moins 5 tests que *vous* avez écrits.

5.1 Remise

Pour la remise, vous devez remettre trois fichiers (`psil.hs`, `tests.psil`, 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, 15% pour les tests, 60% pour le code Haskell (i.e. *s2l*, `check`, et *eval*).
— 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, est 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.