Travail pratique #1

IFT-2035

15 mai 2023

¡¡ 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**.

$\tau ::= \mathsf{Int}$	Type des nombres entiers
$\mid (\tau_1 \ ... \ \tau_n \to \tau)$	Type d'une fonction
$e ::= n$	Un entier signé en décimal
$\mid x$	Référence à une variable
$\mid (e_0 \ e_1 \ ... \ e_n)$	Un appel de fonction (*curried*)
$\mid (\mathsf{fun} \ x \ e)$	Une fonction d'un argument
$\mid (\mathsf{let} \ ((x \ e_1)) \ e_2)$	Ajout d'une variable locale
$\mid (\ : e \ \tau)$	Annotation de type
$\mid + \mid - \mid * \mid / \mid \mathsf{if0}$	Opérations arithmétiques
$d ::= (\mathsf{dec} \ x \ \tau)$	Déclaration de variable
$\mid (\mathsf{def} \ x \ e)$	Définition de variable
$p ::= d_1 \ ... \ d_n$	Programme

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 qu'avec la syntaxe de style Lisp, les parenthèses sont généralement 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 forme let est utilisée pour donner un nom à une définition locale. Elle n'autorise pas la récursion. Exemple :

$$\begin{array}{l}(\mathsf{let} \ ((x \ 2)) \\ \ \ (+ \ x \ 3))\end{array} \quad \rightsquigarrow^* \quad 5$$

Un programme Psil est une suite de déclarations. Les déclarations ont généralement la forme (def x e) qui défini une nouvelle variable globale x, mais une telle définition peut aussi être précédée comme en Haskell d'une déclaration du type de x avec (dec x τ). Cette déclaration est indispensable si x est définie récursivement (i.e. a besoin de faire référence à elle-même) :

```
(dec infloop (Int → Int))
(def infloop (fun x (infloop (+ x 1))))
```

2.1 Sucre syntaxique

Les fonctions n'ont qu'un seul argument : la syntaxe offre la possibilité de passer plusieurs arguments, mais ces arguments sont passés par *currying*. Plus précisément, les équivalences suivantes sont vraies pour les expressions :

$$\begin{array}{rcl}(e_0 \ e_1 \ e_2 \ ... \ e_n) & \iff & (..((e_0 \ e_1) \ e_2)... \ e_n) \\ (\tau_1 \ ... \ \tau_n \to \tau) & \iff & (\tau_1 \to ...(\tau_n \to \tau)..)\end{array}$$

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

$$\frac{\Gamma \vdash e_1 \Rightarrow (\tau_1 \to \tau_2) \qquad \Gamma \vdash e_2 \Leftarrow \tau_1}{\Gamma \vdash (e_1\ e_2) \Rightarrow \tau_2} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash (\mathsf{fun}\ x\ e) \Leftarrow (\tau_1 \to \tau_2)}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \Gamma, x{:}\tau_1 \vdash e_2 \Rightarrow \tau_2}{\Gamma \vdash (\mathsf{let}\ ((x\ e_1))\ e_2) \Rightarrow \tau_2}$$

FIGURE 2 – Règles de typage

Vous allez utilisez ces équivalences dans la première partie du code qui va convertir le code d'entrée du format générique Sexp au format "Lambda", un format beaucoup plus pratique pour implanter la suite, i.e. la vérification des types et l'évaluation.

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 deux jugements $\Gamma \vdash e \Leftarrow \tau$ et $\Gamma \vdash e \Rightarrow \tau$ qui disent 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 auxquelles e peut faire référence.

Ce genre de typage est dit "bidirectionnel" : $\Gamma \vdash e \Leftarrow \tau$ est utilisé lorsqu'on connaît déjà le type attendu de e et qu'on veut donc vérifier que e a effectivement ce type, et on utilise $\Gamma \vdash e \Rightarrow \tau$ lorsqu'au contraire on ne connaît pas à priori le type de e et donc la règle est sensée "synthétiser" le type τ en analysant e.

Cette approche permet de réduire un peu la quantité d'annotations de types pour le programmeur Psil, en profitant de manière "opportuniste" de l'information de typage disponible, tout en étant beaucoup plus simple à implanter qu'une inférence de types comme celle de Haskell.

Il manque les règles de typage des opérations arithmétiques, car elles sont 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 seulement les entiers et les fonctions.

Les règles de réductions primitives (β) sont les suivantes :

$$\begin{array}{rcl} ((\mathsf{fun}\ x\ e)\ v) & \rightsquigarrow & e[v/x] \\ (\mathsf{let}\ ((x\ v))\ e) & \rightsquigarrow & e[v/x] \end{array}$$

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

$$\begin{array}{rcl} (+\ n_1\ n_2) & \rightsquigarrow & n_1 + n_2 \\ (-\ n_1\ n_2) & \rightsquigarrow & n_1 - n_2 \\ (*\ n_1\ n_2) & \rightsquigarrow & n_1 \times n_2 \\ (/\ n_1\ n_2) & \rightsquigarrow & n_1 \div n_2 \\ (\mathsf{if0}\ n_1\ n_2\ n_3) & \rightsquigarrow & \begin{cases} n_2 & \text{si } n_1 = 0 \\ n_3 & \text{sinon} \end{cases} \end{array}$$

Donc il s'agit d'une variante du λ-calcul, sans grande surprise. La portée est statique et l'ordre d'évaluation est 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 *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 élaboraion, termine l'analyse syntaxique et commence la compilation, en transformant cet arbre en un vrai arbre de syntaxe abstraite dans la représentation appellée "Lambda" (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 $\ldots \Longleftrightarrow \ldots$).
3. Une troisième phase, appelée *check/synth*, vérifie que le code est correctement typé et en trouve son type.
4. Finalement, une fonction *eval* procède à l'évaluation des expressions par interprétation, et *process_decl* l'utilise pour évaluer toutes les déclarations du programme.

Votre travail consiste à compléter ces phases.

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 alors à la fin de la ligne.

3.2 La représentation intermédiaire "Lambda"

Cette représentation intermédiaire est le cœur de notre implémentation de Psil, et représente 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.

Elle est définie par les types suivants :

```
data Lexp = Lnum Int            -- Constante entière.
          | Lvar Var            -- Référence à une variable.
          | Lhastype Lexp Ltype -- Annotation de type.
          | Lapp Lexp Lexp      -- Appel de fonction, avec un argument.
          | Llet Var Lexp Lexp  -- Déclaration de variable locale.
          | Lfun Var Lexp       -- Fonction anonyme.
          deriving (Show, Eq)
```

```
data Ltype = Lint
           | Larw Ltype Ltype    -- Type "arrow" des fonctions.
           deriving (Show, Eq)

data Ldec = Ldec Var Ltype       -- Déclaration globale.
          | Ldef Var Lexp        -- Définition globale.
          deriving (Show, Eq)
```

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 "élabore" une expression de type *Sexp* en *Lexp*.

check et *synth* sont les fonctions qui vérifient et synthétisent (infèrent) 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 évalue sur toutes les déclarations trouvées dans ce fichier.

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

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

psil.hs:310:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘eval’:
        Patterns not matched:
```

```
            _ (Lhastype _ _)
            _ (Lapp _ _)
            _ (Llet _ _ _)
            _ (Lfun _ _)
    |
310 | eval _venv (Lnum n) = Vnum n
    | ^^^^^^^^^^^^^^^^^^^^^^^^^^^...

psil.hs:322:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘process_decl’:
        Patterns not matched: ((_, _), Just (_, _), _) (Ldef _ _)
    |
322 | process_decl (env, Nothing, res) (Ldec x t) = (env, Just (x,t), res)
    | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
Ok, one module loaded.
*Main> run "exemples.psil"
  2 : Lint
  <fonction> : Larw Lint (Larw Lint Lint)
  *** Exception: Expression Psil inconnue: (+ 2)
CallStack (from HasCallStack):
  error, called at psil.hs:223:10 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*, *eval*, ... Je recommande de le faire en avançant 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 et qui devraient tester différents recoins de Psil.

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. élaboration, vérification des types, et évaluation).
— 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.