Travail pratique #1

IFT-2035

15 mai 2021

¡¡ Dû le 30 mai 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 `frontal.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**.

$\tau ::=$	Int	Type des nombres entiers
	| Bool	Type des booléens
	| $(\tau_1 \ ... \ \tau_n \to \tau)$	Type d'une fonction
	| (Tuple $\tau_1 \ ... \ \tau_n$)	Type d'un tuple
$e ::=$	n	Un entier signé en décimal
	| $+ \mid - \mid * \mid / \mid \leq \mid ...$	Opérations prédéfinies
	| x	Une variable
	| (hastype $e \ \tau$)	Annotation de type
	| (call $e_0 \ e_1 \ ... \ e_n$)	Un appel de fonction (*curried*)
	| (fun $x_1 \ ... \ x_n \ e$)	Une fonction ; les arguments sont *curried*
	| (let $d_1 \ ... \ d_n \ e$)	Ajout de déclarations locales
	| (if $e_1 \ e_2 \ e_3$)	Expression conditionelle
	| (tuple $e_1 \ ... \ e_n$)	Construction de tuple
	| (fetch $e_1 \ (x_1 \ ... \ x_n) \ e_2$)	Lecture d'un tuple
$d ::=$	$(x \ e)$	Déclaration de variable
	| $(x \ \tau \ e)$	Déclaration de variable avec son type
	| $(x \ (x_1 \ \tau_1) \ ... \ (x_n \ \tau_n) \ \tau \ e)$	Déclaration de fonction avec son type

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. Un programme Psil est une expression, et dans notre fichier de test, nous aurons une séquence d'expressions, qui sont évaluées indépendamment l'une de l'autre. À remarquer que, comme toujours avec la syntaxe de style Lisp, les parenthèses sont *significatives*.

La forme let est utilisée pour donner des noms à des définitions locales. Exemple :

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

Les définitions d'un let peuvent être mutuellement récursives. Exemple :

```
(let (even (x Int) Bool
       (if (call = x 0) true (call odd (call - x 1))))
     (odd (x Int) Bool                                  ⇝*   false
       (if (call = x 0) false (call even (call - x 1))))
  (call odd 8))
```

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

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

$$\frac{\Gamma' = \Gamma, x_1{:}\tau_1, ..., x_n{:}\tau_n \quad \Gamma' \vdash e \Rightarrow \tau \quad \forall i. \quad \Gamma' \vdash e_i \Rightarrow \tau_i}{\Gamma \vdash (\mathsf{let}\ (x_1\ e_1) ...\ (x_n\ e_n)\ e) \Rightarrow \tau}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow \mathsf{Bool} \quad \Gamma \vdash e_2 \Leftarrow \tau \quad \Gamma \vdash e_3 \Leftarrow \tau}{\Gamma \vdash (\mathsf{if}\ e_1\ e_2\ e_3) \Leftarrow \tau}$$

$$\frac{\forall i. \quad \Gamma \vdash e_i \Rightarrow \tau_i}{\Gamma \vdash (\mathsf{tuple}\ e_1\ ...\ e_n) \Rightarrow (\mathsf{Tuple}\ \tau_1\ ...\ \tau_n)}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow (\mathsf{Tuple}\ \tau_1\ ...\ \tau_n) \quad \Gamma, x_1{:}\tau_1, ..., x_n{:}\tau_n \vdash e_2 \Leftarrow \tau}{\Gamma \vdash (\mathsf{fetch}\ e_1\ (x_1\ ...\ x_n)\ e_2) \Leftarrow \tau}$$

FIGURE 2 – Règles de typage

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 ce n'est que du sucre syntaxique pour des définitions et des appels en forme *curried*. 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 :

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

De plus, la syntaxe des déclarations utilise elle aussi du sucre syntaxique, et elle est régie par les équivalences suivantes sur les déclarations :

$$\begin{array}{rcl} (x\ \tau\ e) & \iff & (x\ (\mathsf{hastype}\ e\ \tau)) \\ (x\ (x_1\ \tau_1)\ ...\ (x_n\ \tau_n)\ \tau\ e) & \iff & (x\ (\tau_1\ ...\ \tau_n \to \tau)\ (\mathsf{fun}\ (x_1\ ...\ x_n)\ e)) \end{array}$$

2.2 Sémantique statique

Une des différences les plus notoires entre Lisp et Psil est que Psil est typé statiquement. Les règles de typage sont divisées en 2 *jugements* qui s'utilisent dans deux sens différents : $\Gamma \vdash e \Leftarrow \tau$ est la jugement de *vérification* qui s'assure que l'expression e a bien le type τ, dans le cas où on connaît τ d'avance, alors que $\Gamma \vdash e \Rightarrow \tau$ est la règle de *synthèse* qui vérifie que e est bien typé et en infère son type τ. Dans chacune de ces règles, Γ représente le contexte de typage, c'est

à dire qu'il contient le type de toutes les variables auxquelles e a le droit de faire référence.

Cette division en deux jugements permet de réduire la quantité d'annotations de types, tout en gardant un système beaucoup plus simple que celui de Haskell. Il y a deux règles qui permettent de passer d'un jugement à l'autre : la règle du hastype qui permet d'aider le système en lui fournissant explicitement l'information de type manquante ; et la règle sans nom (en haut à droite) qui s'utilise là où il y a de l'information de type redondante et qu'il faut par conséquent vérifier que les différentes sources d'information sont en accord.

Il n'y a pas de règles de typage pour les opérations prédéfinies, car elles sont simplement des "variables prédéfinies" qui sont donc incluses 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 booléens, les fonctions, et les tuples.

Les règles d'évaluation fondamentales sont les suivantes :

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

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 complètement 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, chacune des différentes primitives vient avec sa ou ses règles de réduction telles que :

$$\begin{array}{rcl} (\mathsf{if}\ v\ e_t\ e_e) & \rightsquigarrow & \begin{cases} e_e & \text{si } v = \mathsf{true} \\ e_t & \text{si } v = \mathsf{false} \end{cases} \\ (\mathsf{tuple}\ v_1\ ...\ v_n) & \rightsquigarrow & [v_1\ ...\ v_n] \\ (\mathsf{fetch}\ [v_1\ ...\ v_n]\ x_1\ ...\ x_n\ e) & \rightsquigarrow & e[v_1, ..., v_n/x_1, ..., x_n] \\ (+\ n_1\ n_2) & \rightsquigarrow & n_1 + n_2 \\ (-\ n_1\ n_2) & \rightsquigarrow & n_1 - n_2 \\ (*\ n_1\ n_2) & \rightsquigarrow & n_1 \times n_2 \\ (/\ n_1\ n_2) & \rightsquigarrow & n_1 \div n_2 \end{array}$$

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

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 "syntax expression") dans le code. C'est une sorte de syntaxe abstraite générique (cela s'apparente en fait à XML ou JSON).représentation est aussi utilisée pour les
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 appelée *Lexp* (pour "lambda expression"). 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. les règles de la forme ... $\Longleftrightarrow$ ...), et doit faire quelques ajustements supplémentaire.
3. Une troisième phase, appelée *infer*, vérifie que le code est correctement typé et infère son type.
4. 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 des morceaux simples des 3 autres. Votre travail consistera à compléter *s2l*, *infer*, *check*, et *eval*.

3.1 Analyse lexicale et syntaxique

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, 'Emacs' 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*

Dans cette représentation, +, -, ... sont simplement des variables prédéfinies, et le sucre syntaxique n'est plus disponible ; entre autres cela signifie que les appels de fonctions ne prennent plus qu'un seul argument.

Elle est définie par le type :

```
data Ltype = Lint                    -- Int
           | Lboo                    -- Bool
           | Larw Ltype Ltype        -- τ1 → τ2
           | Ltup [Ltype]            -- tuple τ1...τn

data Lexp = Lnum Int                 -- Constante entière.
          | Lvar Var                 -- Référence à une variable.
          | Lhastype Lexp Ltype      -- Annotation de type.
          | Lcall Lexp Lexp          -- Appel de fonction.
          | Lfun Var Lexp            -- Fonction anonyme.
          | Llet [(Var, Lexp)] Lexp -- Déclarations locales.
          | Lif Lexp Lexp Lexp       -- Expression conditionelle.
          | Ltuple [Lexp]            -- Construction de tuple.
          | Lfetch Lexp [Var] Lexp   -- Lecture d'un tuple.
```

Cette représentation est au cœur de ce travail : la fonction *s2l* la génère, *infer* l'analyse, et *eval* la consomme.

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, et *Ltype* le type de leurs types.

s2l est la fonction qui transforme une expression de type *Sexp* en *Lexp*.
infer est la fonction qui infère (et vérifie) le type d'une expression.
tenv0 est l'environnement de typage initial.
Value est le type du résultat de l'évaluation d'une expression.
env0 est l'environnement initial.
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. En première approximation, elle correspond à

$$eval\ env0\ (s2l\ (readSexp\ \langle code \rangle))$$

sexpOf, *lexpOf*, *typeOf*, *valOf* sont des fonctions auxiliaires qui peuvent vous être utile dans la boucle interactive pour tester votre code : elles prennent un code Psil en argument et le passent dans la partie correspondante de votre code.

Voici ci-dessous un exemple de session interactive sur une machine GNU/Linux, pour lancer le code fourni :

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

psil.hs:289:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘eval2’:
        Patterns not matched:
            _ (Lcall _ _)
            _ (Lfun _ _)
            _ (Llet _ _)
            _ (Lif _ _ _)
            ...
    |
289 | eval2 _    (Lnum n) = \_ -> Vnum n
    | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...

psil.hs:318:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘infer’:
        Patterns not matched:
            _ (Lhastype _ _)
            _ (Lcall _ _)
            _ (Llet _ _)
            _ (Ltuple _)
    |
318 | infer _ (Lnum _) = Lint
    | ^^^^^^^^^^^^^^^^^^^^^^^...
```

```
Ok, one module loaded.
*Main> run "sample.psil"
  2 : Lint
  <fun +> : Larw Lint (Larw Lint Lint)
  *** Exception: Unrecognized Psil expression: (call + 2 4)
CallStack (from HasCallStack):
  error, called at psil.hs:223:10 in main:Main
*Main>
```

Comme vous le voyez, GHCi n'est pas très content : il dit qu'il manque du code (il dit que *eval2* et *infer* sont incomplets), et ce code manquant cause ensuite une erreur quand on appelle *run*. Votre code devra bien sûr corriger ces problèmes.

5 À faire

Vous allez devoir compléter l'implantation de ce langage, c'est à dire compléter les sections marquée `COMPLETER` dans le code source.

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. Vous pouvez aussi modifier le reste du code, si vous le voulez, mais il faudra alors clairement justifier ces modifications dans votre rapport en expliquant pourquoi cela vous a semblé nécessaire.

Vous devez de plus renvoyer un fichier `tests.psil` qui contient de 5 tests supplémentaire de votre création. Pour ce fichier, vous pouvez vous baser sur le fichier `sample.psil` comme point de départ. Vos tests seront évalué sur les critères suivants :

1. Il faut que le code soit valide
2. Il faut qu'il s'exécute correctement sur votre implantation de Psil.
3. Il faut qu'il détecte un maximum d'erreurs d'implantation. Pour cela, vos tests seront exécutés sur diverses implantations incorrectes ou incomplètes de Psil, telles que celles d'autres équipes.

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. Vous pouvez aussi les mettre dans un fichier `psil.tar` si vous préférez. Assurez-vous que le rapport compile correctement sur `frontal.iro`.

6 Détails

— La note sera divisée comme suit : 25% pour le rapport, 60% pour le code Haskell et 15% pour les tests Psil.

— 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 sera 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 généralement sans importance, sauf si votre code utilise un algorithme vraiment particulièrement ridiculement inefficace.