Travail pratique #2

IFT-2035

June 15, 2025

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 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.

Ceux qui ne peuvent pas faire ce travail en groupe 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**.

τ	::= Int	Type des nombres entiers
	$\mid (\tau_1 \ ... \ \tau_n \rightarrow \tau)$	Type d'une abstraction
	$\mid dt$	Nom d'un type algébrique
e	::= n	Un entier signé en décimal
	$\mid x$	Une variable
	$\mid (e_1 \ e_2 \ ... \ e_n)$	Un appel de fonction (*currifié*)
	$\mid$ (abs $((x_1 \ \tau_1) \ ... \ (x_n \ \tau_n)) \ e$)	Une abstraction (*currifée*)
	$\mid$ (def $(d_1 \ ... \ d_n) \ e$)	Ajout de déclarations locales
	$\mid + \mid - \mid * \mid / \mid ...$	Opérations arithmétiques prédéfinies
	$\mid$ (if $e \ e_t \ e_e$)	Expression conditionelle
	$\mid$ (adt $dt \ (a_1 \ ... \ a_n) \ e$)	Définition de type algébrique
	$\mid$ (new $c \ e_1 \ ... \ e_n$)	Appel de constructeur
	$\mid$ (filter $e \ b_1 \ ... \ b_n$)	Filtrage sur donnée
a	::= $c \mid (c \ \tau_1 \ ... \ \tau_n)$	Définition de constructeur
b	::= $(\_ \ e) \mid (c \ e) \mid ((c \ x_1 \ ... \ x_e) \ e)$	Branche de filtrage
d	::= $(x \ e)$	Déclaration non récursive
	$\mid (x \ \tau \ e)$	Déclaration de variable avec son type
	$\mid (x \ (x_1 \ \tau_1) \ ... \ (x_n \ \tau_n) \ \tau \ e)$	Déclaration de fonction

Figure 1: Syntaxe de Psil typé

2 Psil typé statiquement

Vous allez travailler sur l'implantation d'une version typée statiquement du langage du TP2. 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. De même, en accord avec la tradition de Lisp, la notation est toujours de style préfixe, avec l'exception notoire du type des abstractions qui utilise un `->` en avant-dernière position comme "marqueur".

La forme adt est utilisée pour définir de nouveaux types algébriques, similaires à ceux de Haskell. Dans une petite mesure le code Psil du TP1 peut être utilisé tel quel. Exemple:

$$\begin{array}{l}(\text{def } ((x\ 2)\\ \qquad (y\ 3))\\ \ (+\ x\ y))\end{array} \quad \leadsto^* \quad 5$$

Mais dès qu'une fonction est définie, le code est un peu différent car il faut y ajouter des annotations de type pour chaque argument:

$$\begin{array}{l}(\text{def } ((y\ 10)\\ \qquad (\textit{div2}\ ((x\ \text{Int}))\\ \qquad\ (/\ x\ 2)))\\ \ (\textit{div2}\ y))\end{array} \quad \leadsto^* \quad 5$$

$$\frac{}{\Delta;\Gamma \vdash n \ : \ \mathsf{Int}} \qquad \frac{\Delta;\Gamma \vdash e_1 \ : \ (\tau_1 \to \tau_2) \qquad \Delta;\Gamma \vdash e_2 \ : \ \tau_1}{\Delta;\Gamma \vdash (e_1 \ e_2) \ : \ \tau_2}$$

$$\frac{\Gamma(x) = \tau}{\Delta;\Gamma \vdash x \ : \ \tau} \qquad \frac{\Delta;\Gamma, x:\tau_1 \vdash e \ : \ \tau_2}{\Delta;\Gamma \vdash (\mathsf{abs}\ ((x\ \tau_1))\ e) \ : \ (\tau_1 \to \tau_2)}$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma, x_1:\tau_1, ..., x_n:\tau_n \qquad \Delta, \Gamma' \vdash e \ : \ \tau \\ \forall\ d_i = (x_i\ e_i) \text{ on a } \Delta, \Gamma' \vdash e_i \ : \ \tau_i \\ \forall\ d_i = (x_i\ \tau_i\ e_i) \text{ on a } \Delta, \Gamma' \vdash e_i \ : \ \tau_i\end{array}}{\Delta;\Gamma \vdash (\mathsf{def}\ (d_1\ ...\ d_n)\ e) \ : \ \tau}$$

$$\frac{\begin{array}{c}\Delta' = \Delta, dt \mapsto \{c_1 \mapsto (\tau_{11}\ ...), ..., c_n \mapsto (\tau_{n1}\ ...)\} \\ \Delta';\Gamma \vdash e \ : \ \tau\end{array}}{\Delta;\Gamma \vdash (\mathsf{adt}\ dt\ ((c_1\ \tau_{11}\ ...)\ ...\ (c_n\ \tau_{n1}\ ...))\ e) \ : \ \tau}$$

$$\frac{\Delta(dt)(c) = (\tau_1\ ...\ \tau_n) \qquad \forall i. \quad \Delta;\Gamma \vdash e_i \ : \ \tau_i}{\Delta;\Gamma \vdash (\mathsf{new}\ c\ e_1\ ...\ e_n) \ : \ dt}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e \ : \ dt \\ \forall\ b_i = ((c_i\ x_{i1}\ ...\ x_{im})\ e_i) \text{ vérifier:} \\ \Delta(dt)(c_i) = (\tau_{i1}\ ...\ \tau_{im}) \quad \text{et} \quad \Delta;\Gamma, x_{i1}:\tau_{i1}, ..., x_{im}:\tau_{im} \vdash e_i \ : \ \tau\end{array}}{\Delta;\Gamma \vdash (\mathsf{filter}\ e\ (b_1\ ...\ b_n) \ : \ \tau}$$

Figure 2: Règles de typage

De plus new ou filter ne peuvent maintenant utiliser que des constructeurs déclarés auparavant via un type algébrique, e.g.:

```
(adt ListInt
     ((nil:)
      (cons: Int ListInt))
  (def ((even ((xs ListInt))
              (filter xs (nil: 0) ((cons: x xs) (odd xs))))
        (odd ((xs ListInt))
              (filter xs (nil: 1) ((cons: x xs) (even xs)))))
    (odd (new cons: 2 (new cons: 3 (new nil:))))))
```

$\rightsquigarrow^*$ 1

2.1 Sémantique statique

Une des différences les plus notoires entre Lisp et Psil est que Psil est typé statiquement. Les règles de typage sont présentées dans la Figure 2: $\Delta;\Gamma \vdash e \ : \ \tau$ est le jugement de vérification de type qui dit que l'expression e est typée correctement et a le type τ. Il n'y a pas d'*inférence* de types, ici, pour garder le système beaucoup plus simple que celui de Haskell, au coût de l'ajout d'annotations de types pour tous les arguments de fonctions ainsi que pour le type de la valeur de retour des fonctions récursives.

Dans chacune de ces règles, $\Delta;\Gamma$ représente le contexte de typage. Γ est la partie du contexte qui contient le type de toutes les variables auxquelles e a le droit de faire référence, c'est à dire, les variables déjà définies lorsqu'on arrive à e. Δ fait la même chose pour les types qui ont été déjà définis avec adt.

Les quatre premières règles, pour les constantes numériques, les applications de fonctions, les variables, et les fonctions, sont les règles habituelles du lambda calcul typé.

La règle du def est un peu délicate: l'environnement Γ' a besoin du type de chaque variable définie dans le def, mais pour les déclarations de la forme $(x\ e)$, ce type n'est fourni que par l'intermédiaire de $\Delta';\Gamma' \vdash e\ :\ \tau$ qui lui même a besoin de Γ', donc il y a une dépendance circulaire problématique. En pratique, on peut résoudre ce problème de différentes manières. Par exemple, on peut utiliser $\Delta';\Gamma'' \vdash e\ :\ \tau$ où Γ'' est un contexte plus simple dans lequel on donne aux variables sans annotation de type un type bidon invalide (que l'on nomme volontiers $\bot$).

Les trois règles restantes sont celles du adt, du new, et du filter. La règle du adt rajoute simplement dans Δ le type nouvellement défini (une information qui sera utilisée ensuite dans le filter et le new). La règle du new s'assure que le constructeur existe dans Δ et utilise l'information associée pour vérifier que le nombre et le type des arguments est correct. La règle du filter vérifie d'abord que l'expression que l'on teste est effectivement d'un type algébrique dt, puis elle utilise Δ pour vérifier que le motif de chaque branche est valide (existe pour dt, avec le bon nombre de champs), et vérifie finalement que le corps de chaque branche a le bon type.

À noter qu'il n'y a pas de règles de typage des opérations arithmétiques, car elles sont traitées simplement comme des "variables prédéfinies" qui sont donc incluses dans le contexte Γ initial.

Une partie importante 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 seulement de trouver le type d'une expression mais plutôt de trouver d'éventuelles erreurs de typage, donc il est important de tout vérifier.

2.2 Sémantique dynamique

La sémantique dynamique du langage est la même qu'au TP1 mais la fonction *eval* a changé pour faire une sorte de compilation qui transforme une expression représentée comme une *Lexp* en une fonction Haskell, ce qui élimine une grosse partie du surcoût dû à l'interprétation:

- Plus besoin de tester quelle *Lexp* doit être exécutée.
- Plus besoin de chercher les variables dans l'environnement avec des comparaisons de chaînes de caractères.

3 Implantation

Votre travail consiste en plusieurs éléments:

- Corriger *s2l*: La fonction fournie accepte la syntaxe du TP1. Il faut donc l'adapter aux changements et ajouts de la syntaxe.
- Corriger *Lexp*: Le type *Lexp* est lui aussi encore celui du TP1 et doit de même être adapté aux changements et ajouts de la syntaxe.
- Compléter la fonction de vérification des types *check*.
- Compléter la fonction d'évaluation *eval*.

Vous devez aussi fournir un fichier `tests.psil`, qui contient au moins 6 tests que *vous* avez écrits (avec en commentaire une description de ce qui est testé ainsi que la valeur de retour et le type que vous pensez devraient être renvoyés):

- Ils doivent bien sûr être corrects.
- Ils doivent être suffisamment différents les uns des autres (et des exemples fournis) et exercer différentes parties de l'implémentation, pour détecter des erreurs différentes.
- Votre implantation de Psil doit les exécuter correctement.
- 3 de ces tests devraient signaler une erreur de type.
- Au moins 1 de ces 3 tests devrait être du code qui ne signalerait *pas* d'erreur à l'exécution.

3.1 Remise

Pour la remise, vous devez remettre les fichiers suivants: `psil.hs`, `tests.psil`, et `rapport.tex`. Il sont à remettre 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).

4 Détails

- La note sera divisée comme suit: 10% pour *s2l* et *Lexp*, 30% pour *check*, 20% pour *eval*, 20% pour le rapport, et 20% pour les tests.
- Tout usage de matériel (code, 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 inefficace.