Travail pratique #2

IFT-2035

22 novembre 2023

1 Survol

Ce TP vise à améliorer la compréhension des types et du sucre syntaxique en ajoutant ces ingrédients au code du travail précédent. Les étapes de ce travail sont les suivantes :

1. Lire et comprendre cette donnée. Cela prendra probablement une partie importante du temps total.
2. Lire, trouver, et comprendre les parties importantes du code fourni.
3. Compléter et ajuster le code fourni.
4. 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**.

$\tau ::=$	Int	Type des nombres entiers
	| Bool	Type des booléens
	| $(\tau_1 \ ... \ \tau_n \rightarrow \tau)$	Type d'une fonction
	| (Ref τ)	Type d'une *ref-cell*
$e ::=$	n	Un entier signé en décimal
	| x	Une variable
	| $(: \ e \ \tau)$	Annotation de type
	| (begin e_1 ... e_2)	Séquence d'instructions
	| $(\lambda \ (x_1 \ ... \ x_i) \ e_1 \ ... \ e_n)$	Une fonction avec i arguments
	| $(e_0 \ e_1 \ ... \ e_n)$	Un appel de fonction (*curried*)
	| (ref ! e)	Construction d'une *ref-cell*
	| (get ! e)	Chercher la valeur de la *ref-cell* e
	| (set ! e_1 e_2)	Changer la valeur de la *ref-cell* e_1
	| $+ \mid - \mid * \mid /$	Opérations arithmétiques prédéfinies
	| $< \mid > \mid = \mid <= \mid >=$	Opérations booléennes sur les entiers
	| (if e_1 e_2 e_3)	Si e_1 alors e_2 sinon e_3
	| (let x e_x e_1 ... e_n)	Déclaration locale simple
	| (letrec $(d_1 \ ... \ d_i)$ e_1 ... e_n)	Déclarations locales récursives
$d ::=$	$(x \ e)$	
	| $((x \ (x_1 \ \tau_1) \ ... \ (x_n \ \tau_n)) \ \tau \ e_1 \ ... \ e_n)$	

FIGURE 1 – Syntaxe de Slip

2 Slip Typé

Vous allez travailler sur l'implantation d'une version de Slip avec typage statique. La syntaxe de ce langage est décrite à la Figure 1.

La sémantique dynamique de Slip Typé est la même que celle de Slip.

2.1 Sucre syntaxique

Mis à part l'ajout de la forme $(: \ e \ \tau)$ pour les annotations de types, qui vient alle même avec la nouvelle catégorie syntaxique τ pour écrire ces types, la syntaxe du langage est étendue avec du sucre syntaxique qui offre des manière plus pratiques d'écrire les expressions Slip :

$$
\begin{array}{rcl}
(e_1 \ e_2 \ ... \ e_n) & \simeq & ((e_1 \ e_2) \ ... \ e_n) \\
(\text{begin } e_1 \ ... \ e_n) & \simeq & (\text{let } \_ \ e_1 \ (\text{let } \_ \ e_2 \ (... \ e_n)..)) \\
(\lambda \ xs \ e_1 \ ... \ e_n) & \simeq & (\lambda \ xs \ (\text{begin } e_1 \ ... \ e_n)) \\
(\lambda \ (x_1 \ ... \ x_n) \ e) & \simeq & (\lambda \ x_1 \ ...(\lambda \ x_n \ e)..) \\
(\text{let } x \ e_x \ e_1 \ ... \ e_n) & \simeq & (\text{let } x \ e_x \ (\text{begin } e_1 \ ... \ e_n)) \\
(\text{letrec } ds \ e_1 \ ... \ e_n) & \simeq & (\text{letrec } ds \ (\text{begin } e_1 \ ... \ e_n))
\end{array}
$$

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

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

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

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

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

$$\frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash (\mathsf{ref}\,! \; e) \Rightarrow (\mathsf{ref} \; \tau)} \qquad \frac{\Gamma \vdash e \Rightarrow (\mathsf{ref} \; \tau)}{\Gamma \vdash (\mathsf{get}\,! \; e) \Rightarrow \tau}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow (\mathsf{ref} \; \tau) \qquad \Gamma \vdash e_2 \Rightarrow \tau}{\Gamma \vdash (\mathsf{set}\,! \; e_1 \; e_2) \Rightarrow \tau}$$

FIGURE 2 – Règles de typage

Les déclarations qui peuvent apparaître dans un letrec offrent elles aussi du sucre syntaxique :

$$((x \; (x_1 \; \tau_1) \; ... \; (x_n \; \tau_n)) \; \tau \; e_1 \; ... \; e_n) \quad \simeq \quad (x \; (: \; (\mathsf{lambda} \; (x_1 \; ... \; x_n) \; e_1 \; ... \; e_n) \; (\tau_1 \; .. \; \tau_n \; \to \; \tau)))$$

2.2 Sémantique statique

Une des différences les plus notoires entre Slip et Slip Typé est que Slip Typé 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 le 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 synthétise 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 : 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.

3 Travail

La donnée est une solution du TP1 modifiée légèrement pour y inclure le squelette du TP2 (et changer un peu le format de sortie de run), donc ce qu'il vous reste à faire est :

— Modifier s2l pour accepter les nouveaux éléments de syntaxe.
— Compléter check et synth pour implémenter la vérification des types.
— Ajuster le code exemples.slip pour ce nouveau langage. Au minimum cela signifie d'y ajouter des annotations de types pour que le code soit accepté, mais idéalement il faut aussi modifier le code pour qu'il soit plus facile à lire, en tirant profit des nouveaux éléments de syntaxe.

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 )
Ok, one module loaded.
ghci> run "exemples.slip"
ERREUR: Annotation de type manquante: Llit 2
<function> : Tabs Tint (Tabs Tint Tint)
ERREUR: Annotation de type manquante: Lfuncall (Lid "+") [Llit 2,Llit 4]
ERREUR: Annotation de type manquante: Lfuncall (Labs "x" (Lid "x")) [Llit 2]
ERREUR: Annotation de type manquante: Lfuncall (Lfuncall (Labs "x" ...
ERREUR: Annotation de type manquante: Ldec "true" (Lid "false") (Lite ...
ERREUR: Annotation de type manquante: Lmkref (Llit 5)
ERREUR: Annotation de type manquante: Ldec "c1" (Lmkref (Llit 5)) ...
ERREUR: Annotation de type manquante: Ldec "c1" (Lmkref (Llit 1)) ...
ERREUR: Annotation de type manquante: Lrec [("c1",Lmkref (Llit 1)),...
ERREUR: Annotation de type manquante: Lrec [("a",Lid "+"),("s",Lid "-")] ...
ERREUR: Annotation de type manquante: Lrec [("odd",Labs "x" (Lite ...
ERREUR: Annotation de type manquante: Lrec [("fac",Labs "x" (Lite ...
ghci>
```

Le code que vous soumettez ne devrait pas souffrir d'avertissements, et l'appel à `run` devrait renvoyer la liste des valeurs décrites en commentaires dans le fichier avec leur type.

Vous devez aussi fournir un fichier `tests.slip`, similaire à `exemples.slip` sauf qu'il contient des tests (au moins 5) qui devraient être *refusés* par le vérificateur de type, avec une explication de l'erreur de type pour chacun des tests. Au moins 3 de ces tests doivent contenir du code dont l'exécution par `eval` n'échouerait *pas*.

4 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.

Pour la vérification des types, 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. Pour l'ajout de sucre syntaxique et pour mettre à jour les exemples, par contre vous devrez modifier le code.

4.1 Remise

Pour la remise, vous devez remettre quatre fichiers (`slip.hs`, `exempls.slip`, `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).

5 Détails

— La note sera divisée comme suit : 25% pour le rapport, 60% pour le code (réparti entre les 3 parties : sucre syntaxique, vérification des types, et mise à jour du code Slip), 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.