Travail pratique #2

IFT-2035

17 novembre 2024

1 Survol

Ce TP vise à améliorer la compréhension des types et de la portée statique en modifiant le 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**.

$\tau ::= \mathsf{Num}$	Type des nombres entiers
$\mid \mathsf{Bool}$	Type des booléens
$\mid (\tau_1 \ ... \ \tau_n \to \tau)$	Type d'une fonction
$e ::= n$	Un entier signé en décimal
$\mid x$	Une variable
$\mid (: \ e \ \tau)$	Annotation de type
$\mid (\mathsf{if} \ e \ e_{\mathsf{then}} \ e_{\mathsf{else}})$	Expression conditionelle
$\mid (e_0 \ e_1 \ ... \ e_n)$	Un appel de fonction
$\mid (\mathsf{fob} \ ((x_1 \ \tau_1) \ ... \ (x_n \ \tau_n)) \ e)$	Une fonction
$\mid (\mathsf{let} \ x \ e_1 \ e_2)$	Déclaration locale simple
$\mid (\mathsf{fix} \ (d_1 \ ... \ d_n) \ e)$	Déclarations locales récursives
$\mid + \mid - \mid * \mid / \mid ...$	Opérations prédéfinies
$d ::= (x \ e)$	Déclaration de variable
$\mid (x \ \tau \ e)$	Déclaration typée
$\mid ((x \ (x_1 \ \tau_2) \ ... \ (x_n \ \tau_n)) \ e)$	Déclaration de fonction
$\mid ((x \ (x_1 \ \tau_2) \ ... \ (x_n \ \tau_n)) \ \tau \ e)$	Déclaration complète

FIGURE 1 – Syntaxe de SSlip

2 SSlip : Un Slip typé ssstatiquement

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. Par rapport à Slip, les changements sont :

- l'ajout de la forme $(: \ e \ \tau)$ qui se comporte comme e mais qui de plus annonce que cette expression doit avoir le type τ, ce qui sera vérifié pendant la vérification des types.
- L'ajout de type qui accompagne dorénavant chaque argument de fonction.
- La possibilité d'ajouter une annotation de type aux déclarations.

La sémantique dynamique de SSlip est la même que celle de Slip, modulo le fait qu'il y a maintenant des annotations de types qu'il faut ignorer.

2.1 Sucre syntaxique

La première partie du travail est la même que celle du premier TP, qui est de transformer le code du format *Sexp* à *Lexp*. Vous pourrez en grande partie reprendre votre code (ou celui du solutionnaire) pour ça, mais il faudra l'ajuster à la nouvelle syntaxe et au nouveau sucre syntaxique. Le sucre syntaxique de SSlip est similaire à celui de Slip, mais il y a maintenant plus de cas :

$$\begin{aligned} (x \ \tau \ e) &\iff (x \ (: \ e \ \tau)) \\ ((x \ \mathit{args...}) \ e) &\iff (x \ (\mathsf{fob} \ (\mathit{args...}) \ e)) \\ ((x \ \mathit{args...}) \ \tau \ e) &\iff (x \ (\mathsf{fob} \ (\mathit{args...}) \ (: \ e \ \tau))) \end{aligned}$$

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

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

$$\frac{\Gamma \vdash e_0 \ : \ (\tau_1 \ ... \ \tau_n \rightarrow \tau_r) \qquad \forall i. \ \Gamma \vdash e_i \ : \ \tau_i}{\Gamma \vdash (e_0 \ e_1 \ ... \ e_n) \ : \ \tau_r}$$

$$\frac{\Gamma, x_1 : \tau_1, ..., x_n : \tau_n \vdash e \ : \ \tau_r}{\Gamma \vdash (\mathsf{fob} \ ((x_1 \ \tau_1) \ ... \ (x_n \ \tau_n)) \ e) \ : \ (\tau_1 \ ... \ \tau_n \rightarrow \tau_r)}$$

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

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

FIGURE 2 – Règles de typage de SSlip

2.2 Sémantique statique

Vu que SSlip est typé statiquement, il est accompagné de règles de typage, qui sont présentées à la Figure 2. Le jugement de typage s'écrit $\Gamma \vdash e \ : \ \tau$ et se lit "e a type τ dans l'environnement Γ". Γ contient le type de toutes les variables dont la portée couvre e, i.e. il contient le type de toutes les variables auxquelles e a le droit de faire référence.

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.

La vérification de type est implantée par la fonction *check* :

```
check :: Bool -> TEnv -> Lexp -> Type
```

Cette fonction doit être totale : en cas d'erreur elle doit renvoyer un "type" de la forme "*Terror s*" où *s* est une chaîne de caractères qui décrit l'erreur. Le premier argument, un booléen, prend normalement la valeur True et indique que la fonction doit bien vérifier que l'expression est typée correctement et renvoyer une erreur si ce n'est pas le cas. Par contre si l'argument est False, cela signifie que *check* peut présumer que l'expression est typée correctement et doit simplement trouver son type.

Cet argument booléen est utilisé pour fix. Parmis les règles ci-dessus, la règle du fix nécessite de "deviner" le type des x_i pour construire Γ'. Par exemple avec

un code source comme :

$$\begin{array}{l}(\mathsf{fix}\ (((\mathit{fact}\ (n\ \mathsf{Num}))\\ \qquad\quad (\mathsf{if}\ (>\ \ n\ 0)\ (*\ n\ (\mathit{fact}\ (-\ n\ 1)))\ 1)))\\ \quad ...)\end{array}$$

Pour pouvoir vérifier les types dans le corps de la fonction, il faut connaître le type de *fact* dans l'environnement. Mais les annotations de types ne nous donnent que le type de son argument et non celui de sa valeur de retour. Cependant, on peut trouver le type de la valeur de retour de *fact* en consultant son code.

Donc chaque expression e_i de fix sera d'abord passée à "*check* False" (avec un environnement qui donne un type invalide aux nouvelles variables, vu qu'on ne connaît pas encore leur type), pour essayer de trouver justement le type de chaque e_i, de manière à "deviner" le Γ' qu'on utilise ensuite pour appeler *check* une deuxième fois pour chaque e_i, cette fois-ci avec l'argument True.

3 Optimisation

La troisième partie du travail consiste à implanter une phase d'optimisation du code, qui transforme des expressions de type *Lexp* en *Dexp*. Cette optimisation fait deux changements :

- Effacement des types : après avoir vérifier les types, on a plus besoin de cette information pour exécuter le code, donc on peut effacer toutes les informations relatives aux types.
- De même la loi du renommage-α nous garanti que les noms de variables n'affectent pas l'exécution, et on s'en débarrasse aussi, en les remplaçant par des informations de position dans l'environnement, qu'on appelle des *index de De Bruijn*, du nom du mathématicien qui les a inventés.

3.1 Indexes de De Bruijn

En portée statique, l'environnement d'évaluation à un point du programme a toujours la même forme, avec les même variables placées dans le même ordre (d'ailleurs aussi les mêmes variables, dans le même ordre que dans le Γ utilisé lors de la vérification des types), et seules les valeurs de ces variables peu changer. Donc au lieu de rechercher la variable par son nom à chaque fois, on peut remplacer le nom de la variable par sa position dans l'environnement, ce qui permet un accès beaucoup plus rapide où il n'est plus nécessaire de comparer des noms de variables.

l'utilisation des indexes de De Bruijn élimine les noms de variables dans les déclarations et remplace les références par des nombres qui indiquent la position de la variable à laquelle on fait référence. Concrètement, si on considère un λ-calcul simple, la syntaxe habituelle :

$$e ::= c \mid x \mid \lambda x \to e \mid e_1\ e_2$$

devient

$$e ::= c \mid \#n \mid \lambda \to e \mid e_1\ e_2$$

où $\lambda \to e$ est une fonction qui prend un argument anonyme, et $\#n$ est une référence à le n-ième variable de l'environnement (0 étant la variable la plus récente/proche). Une expression du λ-calcul comme :

$$\lambda x \to \lambda y \to \mathsf{let}\ z = x + y\ \mathsf{in}\ z + x$$

devient :

$$\lambda \to \lambda \to \mathsf{let}\ \#1 + \#0\ \mathsf{in}\ \#0 + \#2$$

On voit ici que les λ ne portent plus le nom de leur argument (vu qu'il est maintenant anonyme), et de même pour le nom de la variable introduite par le let. La référence à y a été remplacée par $\#0$ parce que, à ce point du code, l'environnement contient (par ordre de proximité) y, x et donc y est en position 0. La référence à z a aussi été remplacée par $\#0$ vu qu'à ce point du code l'environnement contient z, y, x et donc z est aussi en position 0. La première référence à x a été remplacée par $\#1$ vu qu'à cet endroit l'environnement contient y, x, alors que la deuxième est remplacée par $\#2$ vu qu'à cet endroit l'environnement contient z, y, x.

4 Travail

La donnée est similaire à celle 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 :

— s2l pour accepter les nouveaux éléments de syntaxe.
— check pour implémenter la vérification des types.
— l2d pour convertir le code aux indexes de De Bruijn.
— eval pour exécuter le code.

Pour s2l et eval je vous recommande de vous inspirer du solutionnaire du TP1.

Vous devez aussi fournir un fichier `tests.sslip`, similaire au `tests.slip` du TP1 sauf qu'il doit contenir au moins 10 tests, dont au moins 5 qui sont typés correctement et au moins 5 qui sont *refusés* par le vérificateur de type, avec une explication de l'erreur de type pour chacun des tests, Au moins 2 des tests dont le type est incorrect doivent contenir du code dont l'exécution par eval n'échouerait *pas*.

4.1 Remise

Pour la remise, vous devez remettre trois fichiers (`sslip.hs`, `tests.sslip`, 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, 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 inefficace.