TP 1

IFT 2035 - Été 2017

23 mai 2017

Le TP1 est à remettre via Studium pour le 13 juin 23h59
Le TP1 se fait en équipe de 2

1 Mini-Haskell

Le TP1 consiste à écrire un petit langage fonctionnel nommé mini-Haskell. Comme son grand frère, mini-Haskell est un langage fonctionnel pur statiquement typé avec évaluation paresseuse. En implantant un langage fonctionnel à l'aide d'un langage fonctionnel, ce TP a pour objectif principal de vous familiariser avec la programmation ... fonctionnelle.

Les principales étapes pour implanter mini-Haskell sont :

1. Implanter un langage minimaliste avec seulement des entiers et des fonctions à une variable.
2. Permettre la définition de fonction à plusieurs variables
3. Ajouter la syntaxe `let` au langage
4. Permettre la définition de types simples (pas de polymorphisme) par l'utilisateur et leur utilisation avec un `case`.
5. Écrire une fonction pour valider les types

Nous verrons plus en détail chacune de ces étapes.

1.1 Syntaxe de mini-Haskell

Pour vous faciliter la tâche, mini-Haskell aura une syntaxe préfixe comme Lisp ou Scheme (S-expression). Ainsi, lorsque vous ajouterez de nouvelles fonctionnalités comme `let` ou `case` vous n'aurez pas à modifier le parseur.

Le fichier Parseur.hs définit le datatype `Sexp` et les fonctions nécessaires pour transformer une chaîne de caractères en `Sexp`. Voici des exemples de la syntaxe.

```
(+ 1 2)

-- Fonction anonyme avec un paramètre x de type Int
```

```
((lambda ((x Int)) x) 4)

-- Let de deux variables Int
(let ((x Int 5)
      (y Int 8))
     (+ x y))

-- Déclaration d'un datatype Bool et ListInt
(data ((Bool True False)
       (ListInt Nil (Cons Int ListInt)))
  (let ((x Bool True)
        (if (Bool -> Bool)
            (lambda ((x Bool))
               (case x ((True False) (False True))))))
     (if x)))
```

1.2 Arbre de syntaxe abstraite (ASA)

Le parseur vous renvoie une S-expression, mais pour mieux vérifier les types et évaluer le programme il est plus utile de transformer la S-expression en un arbre de syntaxe abstraite. Le datatype `Exp` défini dans le fichier Eval.hs est une ébauche de l'ASA. Il est assez complet pour réussir à implanter le langage minimaliste et il faudra le modifier pour y rajouter les syntaxes `let`, `data` et `case`.

La fonction `sexp2Exp` se charge de traduire une S-expression en ASA. Elle est partiellement implantée.

1.3 Implanter un langage minimaliste

Votre premier travail sera de compléter les fonctions `eval` et `typeCheck` du fichier Eval.hs pour pouvoir permettre l'évaluation d'un langage minimaliste.

Il s'agit de pouvoir évaluer des expressions simples comme celles-ci :

```
((+ 1) 2)

((* 1) 2)

((lambda ((x Int)) x) 4)

(((lambda ((x Int))
      (lambda ((y Int)) ((+ x) y)))
    6) 8)
```

Chaque fonction n'a qu'un seul paramètre et l'application de fonction se fait un paramètre à la fois. Les entiers sont les seules données disponibles. Noter que dans la définition d'une fonction avec le mot clé `lambda` chaque paramètre est

toujours accompagné par la déclaration de son type. Cela sera toujours le cas durant le TP 1.

La fonction `sexp2Exp` gère déjà le passage d'une S-expression à un ASA pour le langage minimaliste. Il faut donc seulement implanter les fonctions `eval` et `typeCheck`. La fonction `eval` a pour signature `Env -> Exp -> Value`. La fonction `typeCheck` a pour signature `Tenv -> Exp -> Either Error Type`. Contrairement à `eval`, `typeCheck` retourne soit une erreur ou le type de l'expression. Le programme principal (déjà écrit pour vous) n'évalue pas l'expression si `typeCheck` retourne une erreur. Logiquement, si `typeCheck` retourne un type alors c'est que `eval` ne plantera pas et donc elle retourne directement une valeur.

Voici comment vous y prendre pour valider les types

Variables Les variables peuvent uniquement être déclarées via les paramètres des fonctions. Or les types sont obligatoirement fournis par l'utilisateur. Ainsi, pour savoir le type d'une variable il suffit de regarder dans un environnement qui associe chaque variable à son type déclaré.

Fonctions Une fonction doit être de type $T_1 \to T_2$. T_1 est déjà fourni par l'annotation de type. Pour calculer T_2, il faut ajouter la variable du paramètre et son type dans l'environnement des types et ensuite vérifier le type du corps de la fonction avec ce nouvel environnement.

Applications Une application bien typée est composée d'une fonction de type $T_1 \to T_2$ et d'un unique paramètre de type T_1. L'application elle-même a le type T_2.

1.4 Fonction à plusieurs variables

Il n'est pas très pratique pour l'utilisateur de pouvoir déclarer uniquement des fonctions d'une variable. Vous devez donc maintenant modifier la fonction `sexp2Exp` pour accepter des fonctions de plusieurs variables et également des applications de fonctions avec plusieurs paramètres.

Mais il s'agira uniquement de sucre syntaxique. Votre ASA (datatype `Exp` sera toujours un ASA avec des fonctions d'une variable. Autrement dit, vous devez faire en sorte que lorsque l'utilisateur écrit

```
(foo arg1 arg2 arg3)
```

cela soit équivalent à

```
(((foo arg1) arg2) arg3)
```

De même pour la déclaration de fonction :

```
(lambda ((x Int) (y Int)) body)
```

est équivalent à

```
(lambda ((x Int)) (lambda ((y Int)) body))
```

Ainsi, puisqu'il s'agit de sucre syntaxique vous n'avez pas à modifier votre fonction `eval` et `typeCheck`.

1.5 Ajouter la syntaxe `let` au langage

Pour pouvoir définir plus facilement des variables et surtout pour permettre la récursion mutuelle entre définitions, vous allez ajouter la syntaxe `let`.

Une S-expression correspondant à cette syntaxe commence par le mot clé `let` puis est suivie de déclarations et finalement d'une expression. Chaque déclaration est composée d'un identificateur (variable), d'un type et d'une expression qui servira à calculer la valeur de la variable. Les variables du `let` doivent être toutes différentes. Par exemple :

```
(let ((x Int 5)
      (y Int 8))
     (+ x y))

(let ((x (Int -> Int) (lambda ((x Int)) (+ x x)))
      (y Int 3))
     (x y))
```

Vous devez modifier `sexp2Exp` pour accommoder cette syntaxe. Vous devez également modifier le datatype `Exp` pour prendre en compte cette nouvelle syntaxe dans l'ASA.

Bien sûr il faut également ajuster le code de `eval` et `typeCheck` pour ce nouveau cas. Concernant les règles d'évaluation, les variables sont mutuellement récursives et donc l'ordre de déclaration ne devrait pas affecter le résultat. Le code suivant est tout à fait valide :

```
(let ((x Int y)
      (y Int 8))
     (+ x y))
```

Pour les règles de typage, il faut d'abord s'assurer que le code associé aux variables est bien typé. C'est-à-dire qu'il faut vérifier le type de chaque expression associée à une variable dans les déclarations. Bien sûr, lors de cette vérification l'environnement des types est augmenté de ces mêmes variables que le `let` définit, car tout est mutuellement récursif. Ensuite il faut vérifier le type de l'expression principale du `let`.

1.6 Datatype

Il reste maintenant à ajouter une fonctionnalité importante, la possibilité de déclarer de nouveaux types et constructeurs. Cela sera fait avec le mot clé `data` et les déclarations de nouveaux types devront *toujours* être au niveau global. C'est-à-dire que soit la S-expression débute par `data` ou il n'y a pas d'expression `data` dans le code. Cela correspond à la règle Haskell que les datatypes sont définis au niveau global et va grandement faciliter votre vérification des types.

Une S-expression correspondant à la syntaxe `data` commence par le mot clé `data` puis est suivi de déclarations et finalement d'une expression. Chaque déclaration comprend le nom du nouveau type et une série de constructeurs. Chaque

constructeur, comme en Haskell, doit être unique dans tout le programme et indiquer les types de ces arguments.

```
(data ((Bool True False))
   (let ((x Bool True)) x))

(data ((Bool True False)
       (ListInt Nil (Cons Int ListInt)))
  (let ((x Bool True)
        (if (Bool -> Bool)
              (lambda ((x Bool))
                   (case x ((True False) (False True))))))
      (if x)))
```

La syntaxe `case` permet de déconstruire un type algébrique. Elle est similaire à celle de Haskell mais mise sous S-expression.

```
(data ((ListInt Nil (Cons Int ListInt)))
      (case (Cons 4 Nil)
            ((Nil 0)
             ((Cons x xs) x))))
```

Dans l'exemple ci-haut la deuxième branche du `case` sera sélectionnée et la variable x aura pour valeur 4 et xs aura pour valeur `Nil`.

Pour être un peu plus explicite, faites attention que toutes les branches possèdent un constructeur du même type. Il faut aussi, pour un constructeur, que le nombre de variables dans le `case` soit identique au nombre de paramètres déclarés avec `data`. Dans l'exemple précédent il s'agit de 0 variable pour `Nil` et 2 pour `Cons`.

La règle d'évaluation est la même qu'en Haskell. Votre programme prend la première branche qui est satisfaite. Pour la règle de typage, il faut suivre les étapes suivantes :

1. Vérifier le type de l'expression sur laquelle on applique le filtrage par motif du `case`

2. S'assurer que tous les constructeurs du `case` soient bien les constructeurs du type calculé au point 1.

3. Vérifier le type de chacune des expressions associé à chaque branche. Toutes ces expressions doivent avoir le même type. Pour chaque vérification, il faut ajouter les variables du motif avec leur type dans l'environnement des types. Le type des variables provient de la déclaration `data`

4. Il va sans dire qu'un `case` ne peut que s'utiliser dans la portée du `data` déclarant le type en question.

2 Tests unitaires

Pour vous permettre de valider votre code et également pour lever toutes ambiguïtés qui pourraient se retrouver dans cette donnée, plusieurs tests unitaires sont fournis. Vous pouvez voir ces tests unitaires comme faisant partie de la spécification du langage.

3 Rapport

Vous devez fournir un rapport de 3 pages (ceci ne compte pas la page titre) décrivant votre travail et votre expérience. Si votre programme échoue certains tests unitaires, il faut expliquer pourquoi dans votre rapport et donnez une piste de solution. Indiquer comment vous avez modifié certains datatypes déjà fournis pour implanter mini-Haskell.

4 Évaluation

Des tests unitaires sont disponibles pour tester votre code. L'évaluation du TP 1 se fait en fonction du succès des tests unitaires, de la qualité de votre code et de votre rapport.