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, typé statiquement et de portée lexicale. 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.

La donnée de ce TP1 va d'abord vous présenter le langage mini-Haskell et ensuite définir le travail que vous aurez à faire.

1.1 Syntaxe de mini-Haskell

La syntaxe de mini-Haskell est définie par la grammaire EBNF à la page 2. Elle ressemble beaucoup à la syntaxe de LISP et Scheme puisqu'elle est composée de S-expressions. Il s'agit d'une syntaxe préfixe.

Voici des exemples de S-expression :

```
(+ 1 2)

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

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

Attention les espaces sont importants, c'est-à-dire que `(+1 1)` n'est pas égal à `(+ 1 1)`. Dans le premier cas, il s'agit d'une S-expression avec deux éléments dont le premier est la variable `+1`.

Notez aussi que les déclarations de datatypes sont uniquement autorisées au niveau global. Une expression `data` qui n'est pas au niveau global est une erreur de programmation.

global	$::= e$	Une expression
	$\mid (\textbf{data}\ (t_1 \ldots t_n)\ e)$	Définition de datatypes
e	$::= n$	Un entier signé ou non
	$\mid x$	Une variable
	$\mid (e_1\ e_2 \ldots e_n)$	Une application de fonction
	$\mid (\textbf{lambda}\ ((x_1\ \tau_1) \ldots (x_n\ \tau_n))\ e)$	Définition de fonction
	$\mid (\textbf{let}\ (d_1 \ldots d_n)\ e)$	Déclarations locales
	$\mid (\textbf{case}\ e\ (m_1 \ldots m_n))$	Filtrage par motifs
τ	$::= \text{Int}$	Le type des entiers
	$\mid (\tau_1 \to \ldots \to \tau_n)$	Le type des fonctions
	$\mid dt$	Un type déclaré par **data**
t	$::= (dt\ c_1 \ldots c_n)$	Un type et ses constructeurs
c	$::= x$	Un constructeur sans argument
	$\mid (x\ \tau_1 \ldots \tau_n)$	Un constructeur avec arguments
e	$::= (x\ \tau\ e)$	Une déclaration dans un **let**
m	$::= (cm\ e)$	Une branche du **case**
cm	$::= x$	Un constructeur sans argument
	$\mid (x\ x_1 \ldots x_n)$	Un constructeur avec arguments

Fig. 1 : Syntaxe de mini-Haskell

1.2 Règles de typages de mini-Haskell

Le langage mini-Haskell est typé statiquement. Tel que l'indique la syntaxe, les annotations de types sont obligatoires pour les paramètres des fonctions (`lambda`), les variables localement définies par un `let` et dans la déclaration de nouveaux constructeurs avec la syntaxe `data`. Ces annotations de type sont suffisantes pour vérifier le type d'un programme. Les règles de typage sont assez simples. Elles sont énoncées ci-dessous pour l'ensemble des expressions.

1.2.1 Nombre

Un entier est toujours de type `Int`.

$$\frac{}{\Gamma \vdash n : \text{Int}}$$

1.2.2 Variable

Pour savoir le type d'une variable, il suffit de le vérifier dans l'environnement des types. Puisque chaque déclaration de variables (`lambda` ou `let`) comporte une annotation de types, le type est toujours connu.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

1.2.3 Lambda

Pour vérifier le type d'une expression `lambda`, il faut vérifier le type du corps de la fonction en ajoutant le paramètre et son type dans l'environnement des types.

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\text{lambda } ((x\ \tau_1))\ e) : \tau_1 \to \tau_2}$$

1.2.4 Application de fonction

Pour vérifier le type d'une application de fonction, il faut vérifier que l'opérande est bien une fonction et que l'argument est du bon type.

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2}$$

1.2.5 Let

Le `let` est mutuellement récursif. Pour vérifier son type, il faut d'abord vérifier que chaque définition locale associée à une variable est bien du type déclaré. Ensuite il faut vérifier le type de l'expression de retour.

$$\frac{\Gamma, x_1 : \tau_1 \dots x_n : \tau_n \vdash e_i : \tau_i \qquad \Gamma, x_1 : \tau_1 \dots x_n : \tau_n \vdash e : \tau}{\Gamma \vdash (\text{let } ((x_1\ \tau_1\ e_1) \dots (x_n\ \tau_n\ e_n))\ e) : \tau}$$

1.2.6 data

L'expression `data` augmente l'environnement de type avec de nouveaux types et leurs constructeurs. Il doit uniquement être utilisé au niveau global.

Il faut d'abord vérifier que chaque type est unique et que chaque constructeur est unique. Ensuite, il faut ajouter à l'environnement des types chacun des constructeurs pour chaque type déclaré. Pour un type dt, un constructeur sans paramètres a le type dt et les constructeurs avec m paramètres ont le type $\tau_1 \to ... \to \tau_m \to dt$. En effet, un constructeur avec paramètres est comme une fonction qui va retourner un objet de ce type et un constructeur sans paramètre est comme une constante de ce type.

$$\frac{\dfrac{c_{ij} = (x_{ij}\ \tau_{ij1} ... \tau_{ijk})}{\tau_{ij} = \tau_{ij1} \to ... \to \tau_{ijk} \to dt_i} \qquad \Gamma, c_{11} : \tau_{11} ... c_{nm} : \tau_{nm} \vdash e : \tau}{\Gamma \vdash (\text{data } ((dt_1\ c_{11} ... c_{1n}) ... (dt_n\ c_{n1} ... c_{nm}))\ e) : \tau}$$

1.2.7 case

L'expression `case` permet de déconstruire les types déclarés par l'utilisateur. Pour cela, si l'expression e à déconstruire a un type dt, il faut que chaque branche du `case` soit un constructeur du type dt et qu'il y ait autant de variables pour chaque constructeur qu'il y avait de types dans la déclaration initiale de ce constructeur. Ensuite, l'expression de cette branche est typée en considérant que chaque variable du motif a le type qui lui est associé dans la définition du constructeur. À la fin, il faut que chaque branche retourne le même type.

$$\frac{\Gamma \vdash e : dt \qquad \dfrac{cm_i = (x_i\ x_{i1} ... x_{im})}{\Gamma(x_i) = \tau_{i1} \to ... \to \tau_{im} \to dt} \qquad \Gamma, x_{i1} : \tau_{i1} ... x_{im} : \tau_{im} \vdash e_i : \tau}{\Gamma \vdash (\text{case } e\ ((cm_1\ e_1) ... (cm_n\ e_n))) : \tau}$$

1.3 Règles d'évaluation

Les règles d'évaluation de mini-Haskell sont similaires à celle de Haskell. Toute expression pourra être réduite à une valeur qui est soit un entier, une fonction ou un objet d'un datatype utilisateur.

Vous pouvez donc savoir ce que doit faire votre évaluateur en vous demandant ce que Haskell ferait. Par exemple en Haskell un entier s'évalue à un entier, une expression `lambda` en une fermeture (fonction), une expression `let` retourne la valeur de son expression interne évaluée avec les nouvelles déclarations ajoutées dans l'environnement, l'expression `case` entre dans la branche du premier motif dont le constructeur est identique à celui de l'objet déconstruit, etc.

Vous pouvez aussi vous aider des unittests pour déterminer quelle doit-être la sémantique.

2 Structure de l'évaluateur

Votre travail consiste à écrire un évaluateur pour mini-Haskell. Une partie du travail a déjà été faite pour vous. Cette section explique la structure globale de l'évaluateur et le code qui vous est fourni.

2.1 Structure globale

La structure globale de l'interpréteur est la suivante :

$$\text{String} \xrightarrow{\text{pOneSexp}} \text{Sexp} \xrightarrow{\text{sexp2Exp}} \text{Exp} \xrightarrow{\text{typeCheck}} \text{Exp} \xrightarrow{\text{eval}} \text{Value}$$

L'interpréteur accepte une chaîne de caractères, la transforme en S-expression et ensuite en arbre de syntaxe abstraite. La datatype `Exp` (Expression) représente l'ASA. Ensuite, `typeCheck` vérifie que cette expression est bien typée et seulement si c'est le cas `eval` va évaluer cette expression pour obtenir une valeur représentée par le datatype `Value`.

Le programme principal est composé de deux fonctions, `repl` et `unittests`. Les deux fonctions se trouvent dans le fichier Main.hs que vous n'avez pas à modifier.

2.2 REPL

`repl` est un interpréteur où il est possible d'écrire du code mini-Haskell et d'obtenir la valeur correspondante. `repl` est l'acronyme de read eval print loop. Cet évaluateur n'accepte qu'une S-expression à la fois.

Vous pouvez lancer l'interpréteur en utilisant ghci et en chargeant le fichier Main.hs comme ci-dessous :

```
vincents-mbp :Code Source Archi$ ghci
GHCi, version 8.0.2 : http ://www.haskell.org/ghc/  :? for help
Prelude> :l Main.hs
[1 of 3] Compiling Parseur          ( Parseur.hs, interpreted )
[2 of 3] Compiling Eval             ( Eval.hs, interpreted )
[3 of 3] Compiling Main             ( Main.hs, interpreted )
Ok, modules loaded : Eval, Main, Parseur.
*Main> repl
MiniHaskell>
```

Vous pouvez ensuite entrer une expression après le prompt `MiniHaskell>`. Ceci vous sera utile pour tester votre interpréteur avec des S-expressions de votre choix.

Pour quitter la `repl` vous devez y enter :q comme pour ghci.

2.3 unittests

Pour vous permettre de tester plusieurs S-expression d'un coup, la fonction `unittests` accepte le nom d'un fichier et va évaluer toutes les S-expression à l'intérieur. La syntaxe des unittests est la suivante :

global ::=unittest$^+$	Une série de tests
unittest ::=$(e_1\ e_2)$	Un test unitaire
unittest ::=$(e_1\ Erreur)$	Un test unitaire

Les S-expressions e_1 et e_2 sont évaluées et ensuite les valeurs sont comparées. Si les valeurs sont égales, alors le test réussi sinon le test échoue. La valeur `Erreur` indique qu'on s'attend à ce que l'expression e_1 échoue. S'il n'y a pas d'échec alors le test échoue. L'erreur doit être détectée par `sexp2Exp` ou `typeCheck`.

Vous pouvez appeler `unittests` depuis ghci comme ceci :

```
vincents-mbp :Code Source Archi$ ghci
GHCi, version 8.0.2 : http ://www.haskell.org/ghc/  :? for help
Prelude> :l Main.hs
[1 of 3] Compiling Parseur          ( Parseur.hs, interpreted )
[2 of 3] Compiling Eval             ( Eval.hs, interpreted )
[3 of 3] Compiling Main             ( Main.hs, interpreted )
Ok, modules loaded : Eval, Main, Parseur.
*Main> unittests "unittests.txt"
Test 1 Oups ...
CallStack (from HasCallStack) :
  error, called at ./Eval.hs :142 :17 in main :Eval
Test 2 Oups ...
CallStack (from HasCallStack) :
  error, called at ./Eval.hs :142 :17 in main :Eval
Test 3 Oups ...
CallStack (from HasCallStack) :
  error, called at ./Eval.hs :142 :17 in main :Eval
Test 4 Oups ...
CallStack (from HasCallStack) :
  error, called at ./Eval.hs :142 :17 in main :Eval
Test 5 Oups ...
CallStack (from HasCallStack) :
  error, called at ./Eval.hs :142 :17 in main :Eval
Test 7 : Syntax Error : Ill formed Sexp
Test 10 : Syntax Error : Ill formed Sexp
Test 11 : Syntax Error : Ill formed Sexp
Test 12 : Syntax Error : Ill formed Sexp
```

```
Test 13 : Syntax Error : Ill formed Sexp
Test 14 : Syntax Error : Ill formed Sexp
Test 15 : Syntax Error : Ill formed Sexp
Test 16 : Syntax Error : Ill formed Sexp
Test 19 : Syntax Error : Ill formed Sexp
Ran 19 unittests. 5 OK and 14 KO.
*Main>
```

Évidemment au début la plupart des unittests échouent car vous devez compléter les fonctions du fichier Eval.hs

3 Votre travail

3.1 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.

Vous devez vous référer à la section 1.2 pour les règles de typage.

3.2 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`. Seule la fonction `sexp2Exp` doit être modifiée.

3.3 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))
```

Vous devez vous référer à la section 1.2 pour les règles de typage.

3.4 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.

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`.

Vous devez vous référer à la section 1.2 pour les règles de typage. La règle d'évaluation est la même qu'en Haskell. Votre programme prend la première branche qui est satisfaite. Il vous faudra également modifier le datatype Type du fichier Eval.hs

4 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.

5 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.

6 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.