Typeur pour un langage lambda-calcul

Nour TOUNSI

November 19, 2023

1 Introduction

Ce projet consiste à élaborer un typeur et un évaluateur pour un lamda-calcul élargi, intégrant des fonctionnalités telles que les opérations arithmétiques simples, la manipulation de listes, de chaînes de caractères et l'utilisation de références pour les variables mutables. L'objectif est de créer un système capable de vérifier la cohérence des types et d'interpréter ces expressions étendues de manière efficace et précise, offrant ainsi une extension fonctionnelle et pratique au lambda-calcul.

2 Typage :

Références: Indiquent la possibilité de réassigner une variable à une autre valeur du même type.
Types polymorphes:Appliquent une abstraction de type générique sur plusieurs types concrets différents.
Entiers et chaînes de caractères: Représentent des types de base pour les valeurs numériques et les séquences de caractères respectivement.
Listes: Définit des listes d'éléments du même type, bien que la liste soit non homogène, autorisant des types différents au sein d'une même liste.
Références : Indiquent la possibilité de réassigner une variable à une autre valeur du même type.
Types polymorphes: Appliquent une abstraction de type générique sur plusieurs types concrets différents.
Flèches entre types: Utilisées pour typer les abstractions en spécifiant le type de l'argument et le type de retour.
Type unitaire (Unit): Représente un type retourné par des instructions ne produisant pas de valeur spécifique, telles que les assertions.
Le type générique (inconnu) : en OCaml est utilisé pour représenter un type qui n'est pas encore déterminé ou qui peut être n'importe quel type. Dans le contexte du lambda-calcul, cela permet de créer des fonctions polymorphes qui peuvent être appliquées à des arguments de différents types sans avoir à spécifier explicitement le type de l'argument.
Les fonctions utilisés pour typer un terme :
substitute-type: Cette fonction a pour objectif de réaliser le remplacement d'un type spécifique par un autre au sein de la représentation des types du langage. Ce processus de substitution s'opère en explorant la structure du type fourni (‘t‘) et en appliquant la substitution selon des conditions spécifiques. En somme, cette fonction assure la cohérence et l'intégrité du système de types en manipulant et en modifiant la structure des types, permettant ainsi le remplacement d'une variable spécifiée par un autre type donné. Son rôle principal est de vérifier si le premier argument représente un surtype du deuxième argument, et le cas échéant, elle renvoie ce dernier.
La fonction principale genere des equations de typage à partir d'un terme :La fonction ‘genere-equa‘ prend en argument un terme et un type cible, et génère un ensemble d'équations de type en effectuant un pattern matching sur le terme.Dans ce contexte, le pattern matching est utilisé pour décomposer le terme en ses sous-termes et générer des équations de type pour chacun d'eux.

L'environnement global de typage est utilisé pour stocker les types des variables pendant ce processus. Lorsque ‘genere-equa‘ rencontre une variable, elle peut regarder dans cet environnement pour trouver son type. Si le type de la variable est le même que le type cible, ou un sous-type de celui-ci, alors l'équation est satisfaite.

3 Reduction :

Cette fonction est conçue pour parcourir et réduire les termes du lambda-calcul en appliquant des règles de réduction spécifiques à chaque cas rencontré, tout en respectant une limite de temps pour éviter des calculs infinis. Elle assure ainsi le processus d'évaluation des expressions du lambda-calcul en appliquant les réductions nécessaires selon les règles définies. Elle parcourt les différents constructeurs du terme en utilisant un pattern matching pour gérer les applications d'abstractions et les opérations primitives, remplaçant les arguments par leurs valeurs dans les termes.

4 Inference :

Cette fonction prend une expression du lambda-calcul et génère une équation de type. Elle utilise un algorithme d'unification pour résoudre cette équation et déterminer le type de l'expression. Si l'unification échoue, cela signifie que l'expression n'est pas typable

5 Jeux de tests utilisés :

Beta réduction pour I = $\lambda x.x$
Inférence $deid = \lambda x.x$
Beta réduction$pourK = \lambda x.\lambda y.x$
Inférence $deK = \lambda x.\lambda y.x$
Beta réduction $pourKI$
Inférence de KI
Beta réduction pour KII
Inférence de KII
Beta réduction $pourS = \lambda x.\lambda y.\lambda z.xzyz$
Inférence $deS = \lambda x.\lambda y.\lambda z.xzyz$
Beta réduction pour NAT $= (\lambda x.(x + 1)) * 5)$
Beta réduction pour NAT1 $= (\lambda x.(x + x))$
Inference pour NAT1 $= (\lambda x.(x + x))$
Beta réduction pour NAT2 $= (NAT1 * I)$
Inference pour NAT2 $= (NAT1 * I)$
Beta réduction pour Omega $= (\lambda x.xx)(\lambda y.yy)$
Inference pour Omega $= (\lambda x.xx)(\lambda y.yy)$
Beta réduction pour ifzero
Inférence de ifzero
Beta réduction pour ifempty
Inférence de ifempty
Inférence de ref
$- Expression$: '$IFZERO(Ref(ref(Int3)), Int5, Int2)$'
Inférence de r
$- Expression$: $App(Abs("x", Var"x"), Ref(ref(Int2)))$'

6 Remarques :

Certaines parties de ce code sont issues de discussions avec mes collègues Kismath Adeleke et Rayenne Guiassa
Pour compiler le fichier :
```
1  ocamlc -o projet TAS.ml
```
Puis exécuter le programme :
```
3  ./projet
```

7 Résultats des tests :

```
================================ Lambda Calculus ================================
Identity (I = λx.x) - Beta Reduction :
(fun x -> x)
Inference of Identity (I = λx.x) :
(fun x -> x) ***TYPABLE*** avec le type (T2 -> T2)
K Function (K = λx.λy.x) - Beta Reduction :
(fun x -> (fun y -> x))
Inference of K Function (K = λx.λy.x) :
(fun x -> (fun y -> x)) ***TYPABLE*** avec le type (T6 -> (T5 -> T6))
KI Reduction :
(fun T3 -> (fun T1 -> T1))
Inference of KI :
(fun T3 -> (fun T1 -> T1)) ***TYPABLE*** avec le type (T7 -> (T10 -> T10))
KII Reduction :
(fun T5 -> T5)
Inference of KII :
(fun T5 -> T5) ***TYPABLE*** avec le type (T12 -> T12)
S Function (S = λx.λy.λz.xzyz) - Beta Reduction :
Inference of S Function (S = λx.λy.λz.xzyz) :
(fun x -> (fun y -> (fun z -> ((x z) (y z))))) ***TYPABLE*** avec le type ((T21 -> (T19 -> T18)) -> ((T21 -> T19) -> (T21 -> T18)))
NAT1 (NAT1 = (λx.(x+1))*3) - Beta Reduction :
Inference of NAT1 (NAT1 = (λx.(x+1))*3) :
((fun x -> (x + 1)) 5) ***TYPABLE*** avec le type Nat
NAT2 (NAT2 = λx.(x+x)) - Beta Reduction :
(fun x -> (x + x))
Inference of NAT2 (NAT2 = λx.(x+x)) :
(fun x -> (x + x)) ***TYPABLE*** avec le type (Nat -> Nat)
NAT3 (NAT3 = NAT2 * I) - Beta Reduction :
((fun x -> x) + (fun x -> x))
Inference of NAT3 (NAT3 = NAT2 * I) :
((fun x -> (x + x)) (fun x -> x)) ***PAS TYPABLE*** : type flèche non-unifiable avec Nat
Omega (ω = (λx.xx)(λy.yy)) - Beta Reduction :
((fun y -> (y y)) (fun y -> (y y)))
Inference of Omega (ω = (λx.xx)(λy.yy)) :
((fun x -> (x x)) (fun y -> (y y))) ***PAS TYPABLE*** : occurrence de T30 dans (T30 -> T29)
================================ PCF ================================
Reduction for IFZERO :
1
Inference of IFZERO :
(ifzero (10 - 3) then 4 else 1) ***TYPABLE*** avec le type Nat
Reduction for IFEMPTY :
Inference of IFEMPTY :
(ifempty (tete [10]) then 3 else (tete [])) ***PAS TYPABLE*** : Les types ne sont pas les mêmes
============================== Imperative Traits ==============================
Inference of EX_REF :
(ifzero (ref 3) then 5 else 2) ***PAS TYPABLE*** : type entier non-unifiable avec ref T54
Inference of R :
((fun x -> x) (ref 2)) ***TYPABLE*** avec le type ref Nat
```